Efficient Real-Time Shading with Many Lights

Olsson, Ola ; Billeter, Markus ; Persson, Emil

Abstract: Using many lights in real-time applications has been an important goal for many years. The games industry in particular has strived to increase the number of lights to provide enhanced visual quality and realism. Today, high-end games often make use of hundreds of lights in each frame, and this is likely to be pushed further in the future. The ability to efficiently manage and shade large numbers of lights brings many possibilities, apart from simply allowing light to be cast from many dynamic objects. In addition, it can support visualizing global illumination solutions, or enable detailed artistic light direction. Thus, efficient real-time shading with many lights, represents a potential for solving many of the problems facing the development of next generation high-end games. To achieve the level of performance needed to make this possible, the way which light management and shading is performed has undergone dramatically development in recent years. Both industry and academia has invested great effort pursuing this goal, which has resulted in a large number of new and sometimes competing techniques. This course presents an in-depth exploration of this topic, starting with background and leading up to state of the art research, including recent results on supporting shadows. The course combines production experience from game developers with the latest research into efficient many-light algorithms for both desktop and mobile hardware.

DOI: https://doi.org/10.1145/2659467.2659475

- The first part then is going to be presented by me.
- In this part we will provide an overview of techniques that have been, and still are, used in real-time shading.
- The main focus will be on clustered shading, as this is the most advanced and efficient technique today.

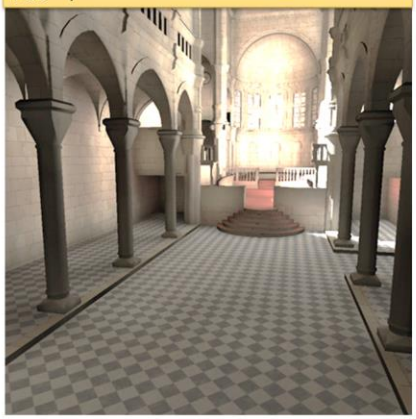
SIGGRAPH Asia 2014, December 03 – 06, 2014, Shenzhen, China.

ACM 978-1-4503-319501/14/12
http://dx.doi.org/10.1145/2659467.2659475

More Lights Please!

- Global Illumination
  - Virtual Point Lights
  - Photon Splatting
- Complex Lights
  - Area / Volume

Photon Splatting (Yao et al. 2010)
739k photons



- So. While I think that you, being a quite advanced audience, have fairly concrete ideas on what many lights can be used for.
- I'll give some illustration as to things that I keep in the back of my head at times like these.
- To the right here we see an image generated using Photon Splatting to visualize the results of global light transport.
- With enough lights, these kind of techniques become possible.

More Lights Please!

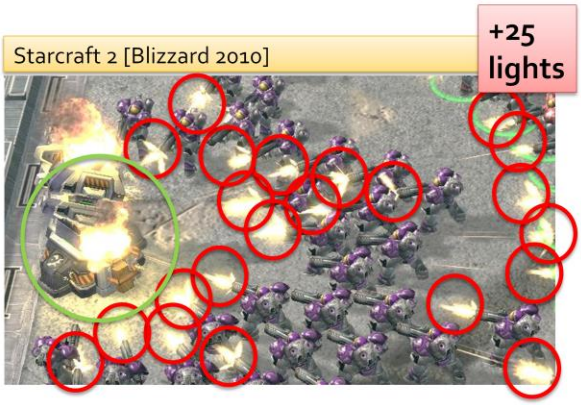
- Artistic Freedom
  - Lighting design.

Need For Speed: The Run, ~2600 lights [1]

- More artistic renderings.
- This scene from Need For Speed: The Run, contains around 2600 lights.
- In movies (animated and otherwise), lots of lights are used to achieve a particular aesthetic.
- This is increasingly going to be the case for games too.
- A benefit to using lights to represent scene illumination, as opposed to baking, is that dynamic geometry is affected

More Lights Please!

- Dynamic Lights
  - Explosions.
  - Particle Effects.
  - Headlights.
  - Torches.
  - Lasers.
  - And so on…



- In games there are a practically limitless number of things that could emit light,
- if we had an efficient way to compute their contributions.
- It's safe to say that current games have not exhausted the possibilities.
- In the starcraft example, there is apparently only a single light <click>.
- However all the muzzle flashes are only additive billboards.

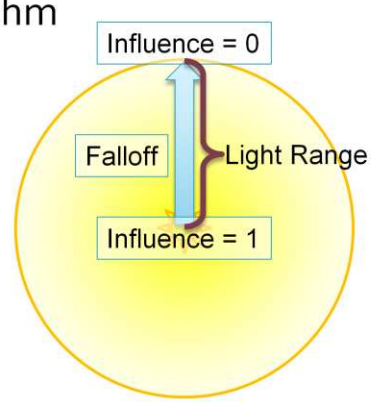
Talk Outline

- Forward Shading
  - Lights and geometry batched together.
- Deferred Shading
  - Decouples lights from geometry.
- Tiled Deferred/Forward Shading
  - Removes bandwidth bottleneck.
- Clustered Deferred/Forward Shading
  - Better robustness, scaling, transparency, MSAA.

- I will show my take on how we can achieve efficient real-time rendering with many lights.
- Our historical and architectural re-cap starts with traditional forward shading, which was predominant until not so long ago.
- Next we look at deferred shading, which was the first widespread technique to bring many lights to games.
- In the previous gen consoles, it was still a popular technique among high end games.
- Tiled deferred shading is now a few years old, and has seen adoption in many modern high-end games.
- Clustered shading, further improves efficiency and scalability over tiled shading.
- We will also discuss forward shading, using both clustered and tiled shading, which allows
- the use of transparency and MSAA, while retaining much of the goooodness with deferred techniques.

Problem Definition

- Real-time shading algorithm
  - Thousands of lights
    - Limited range light
    - No shadows
      - Actually, shadows later!
  - Fully dynamic
    - Lights and Geometry



The algorithms we are going to talk about target thousands of lights in real time, The lights have a limited range, with some falloff which goes to 0 at the boundary. This means that lights are not physical, but this is the normal procedure in games. We also do not consider shadows, which is covered later in the course.

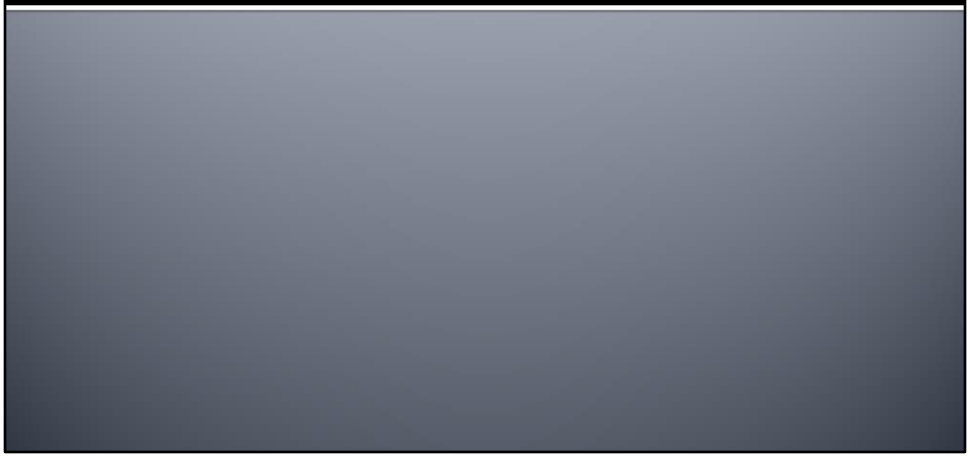
There is no pre-computation so all geometry and lights are allowed to change freely from frame to frame.

Light Assignment

- Question: *Which lights affect what geometry.*
- Solution: *Light Assignment*

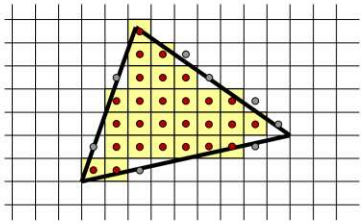
- Most of this problem boils down to working out which lights affect what pieces of geometry.
- We call this ‘light assignment’, but is also know as ‘light culling’.

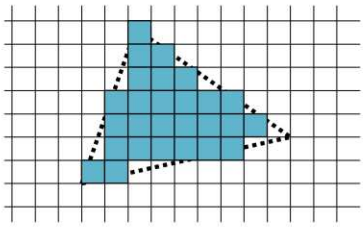
Traditional Forward Shading



(Traditional) Forward Shading

- The traditional approach
  - Single pass
  - Shading in shaders
- Rasterization generates fragments
  - Each fragment is shaded
    - Needs set of lights.
    - Produces a *color*





• The traditional method for real-time shading is called forward shading and used to be the only method during the first decade, or so, of consumer GPUs.
• In this technique, there is only a single pass over the geometry drawing into a frame buffer accumulating the final image.
• Geometry is rasterized, shading is performed and the frame buffer is updated.
• Shading is performed in the fragment shaders (using the like of the example shader we just saw).

Forward Shading

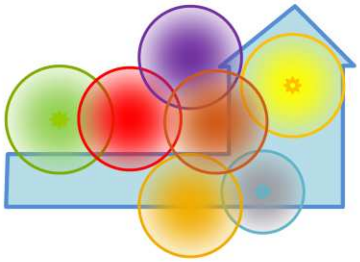
- Which lights to use?
  - Gather before draw call.
- Want minimal set of lights.
  - Small batches.
- Want quick drawing
  - Few batches!
    - ->Large batches.
  - Lower GPU/API overhead.
  - Fewer material changes.

• We want to make sure the fragment shader has access to just the right set of lights, not too many and not too few.
•If we look in the picture we see that none of the lights affect all the fragments produced by each primitive.
• So already in this toy example, assigning the lights per primitive is wasteful.

• Generally gets worse with more geometry and more lights.
• We want to assign per chunk/batch of primitives.
• To minimize number of lights, we want to make sure a batch is small, geometrically.
• But to draw fast, keep the GPU busy and avoid API overhear, we want large batches
• And don't care so much about geometric shape.
• This is a fundamental conflict, where the best we can manage is a compromise. But it is a difficult one to strike.

Forward Shading -Problem Examples

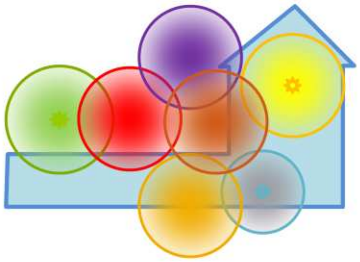
- Few large objects
- Many small lights



• Now, this conflict runs deeper than simply batch size
• The basic problem is that we have to assign lights based on the size of geometry chunks.
•Therefore, on the one hand, we might have a situation with a few large objects, and many small lights

Forward Shading -Problem Examples

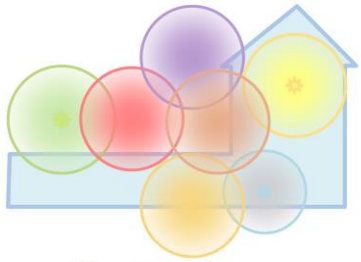
- Few large objects
- Many small lights
- Redundant shading work



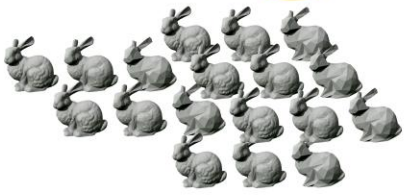
•This leads to lots of wasted effort in shading lights that only affect a portion of the geometry.

Forward Shading -Problem Examples

- Few large objects
- Many small lights
- Redundant shading work



- Many small objects

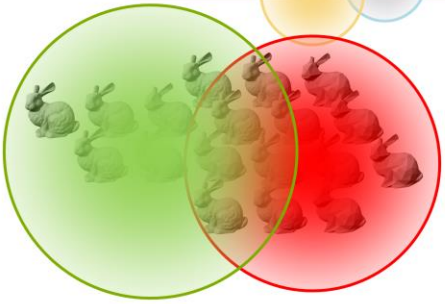


• On the other hand, we may have many small objects.

Forward Shading -Problem Examples

- Few large objects
- Many small lights
- Redundant shading work

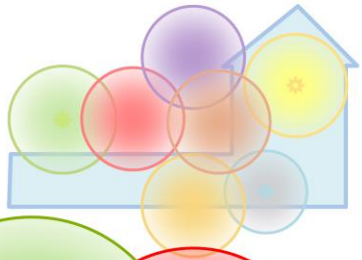
- Many small objects
- Few large lights



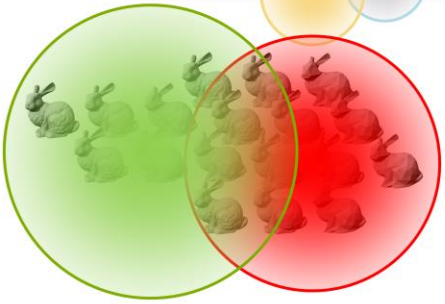
•And a few large lights, in which case we will spend a lot of effort to individually discover that they are affected by the same lights.
•So the conflict runs deeper than just batch size
• In the same scene we might have both situations, which means it is difficult to ensure a good performance balance.

Forward Shading -Problem Examples

- Few large objects
- Many small lights
- Redundant shading work



- Many small objects
- Few large lights
- Redundant light assignment work



•And a few large lights, in which case we will spend a lot of effort to individually discover that they are affected by the same lights.

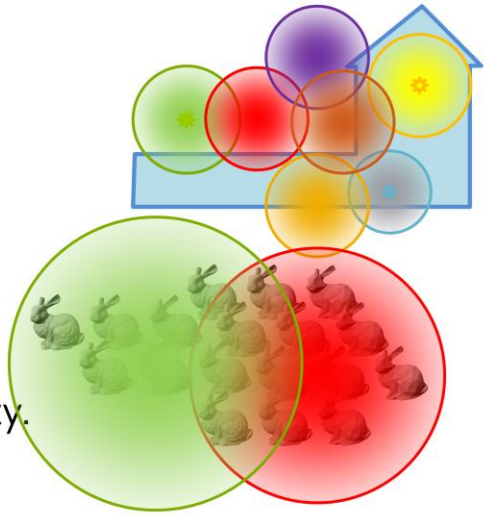
Forward Shading -Problem Examples

- Few large objects
- Many small lights

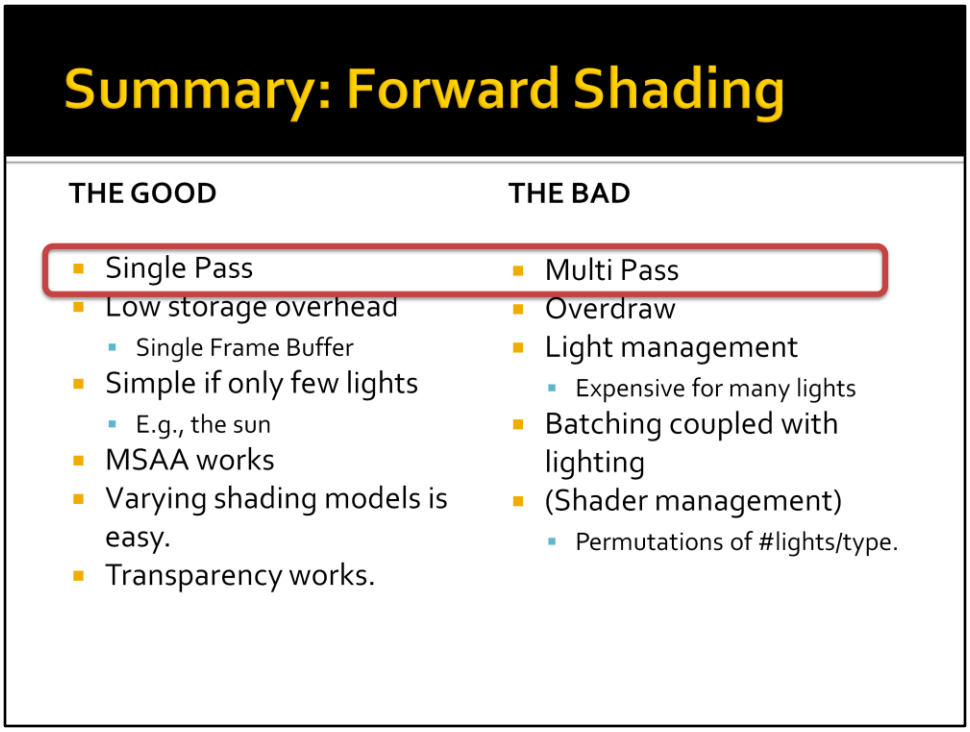
AND

- Many small objects
- Few large lights

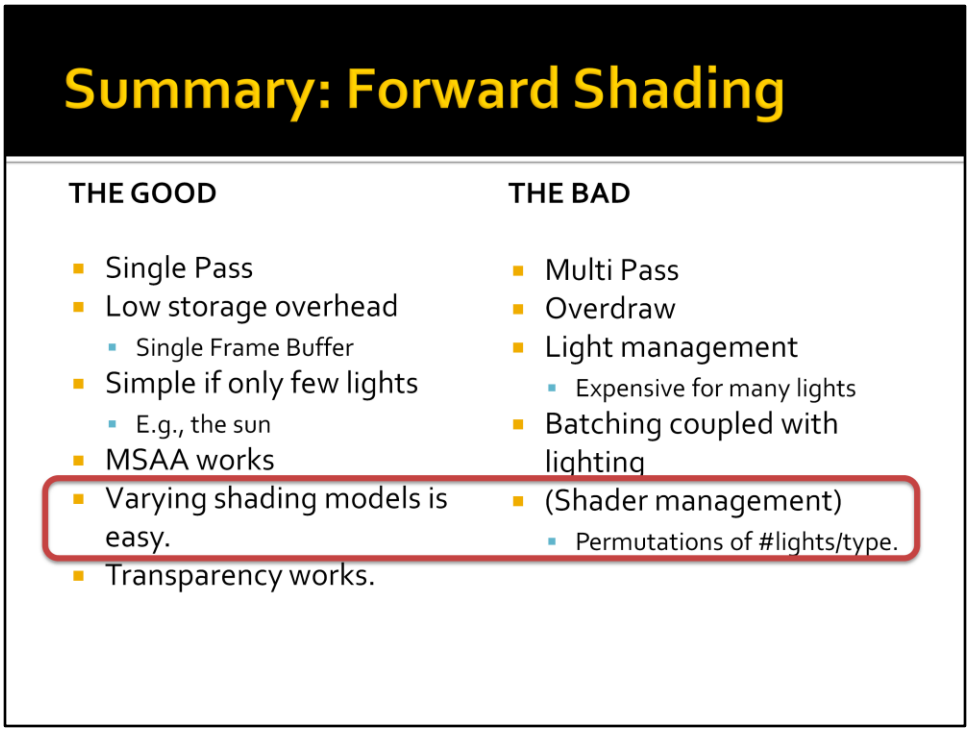
- Large triangles.
- Varying light density.



•So the conflict runs deeper than just batch size…
•And of course, in the same scene we might have both situations, which means it is difficult to ensure a good performance balance.
•Note that the large object could be a single triangle, and in general triangle size and varying light density makes this a very difficult problem to solve well.



- One thing to note here, is that the single pass / multi pass problem here comes from the fact that this is very dependent on the underlying hardware.
    - Many engines targeting Mobile hardware, for example Unity, will do an extra geometry pass per light for the concerned geometry.
    - So if an object is overlapped by 4 lights, it gets drawn 4 times.



- This statement is also not clear cut.
  - Varying shading models are easy, but not if shader management is bogged down with combinatorical explosions.
  - On the other hand, if we do looping over lights in shaders, the problem is greatly mitigated.
  - But, on the third hand this might negatively impact registry usage.

Summary: Forward Shading

THE GOOD

- Single Pass
- Low storage overhead
  - Single Frame Buffer
- Simple if only few lights
  - E.g., the sun
- MSAA works
- Varying shading models is easy.
- Transparency works.

THE BAD

- Multi Pass
- Overdraw
- Light management
  - Expensive for many lights
- Batching coupled with lighting
- (Shader management)
  - Permutations of #lights/type.

- These statements are also dependent on how you implement things, so is not entirely fundamental.

Summary: Forward Shading

THE GOOD

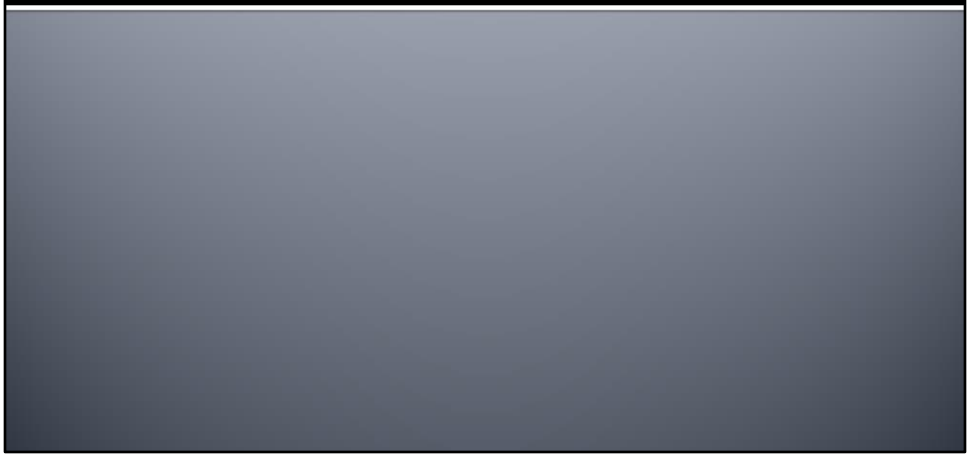
- Single Pass
- Low storage overhead
  - Single Frame Buffer
- Simple if only few lights
  - E.g., the sun
- MSAA works
- Varying shading models is easy.
- Transparency works.

THE BAD

- Multi Pass
- Overdraw
- Light management
  - Expensive for many lights
- Batching coupled with lighting
- (Shader management)
  - Permutations of #lights/type.

- Fortunately, the problem with batching is *almost* an inherent problem.

Traditional Deferred Shading



(Traditional) Deferred Shading

- Modern form introduced around 1990 [Saito90]
  - Term coined later by Tebbs et al [Tebbs92]
  - Consumer GPUs in 2004 [Hargreaves04]
  - Games around 2005 [Shishkovtsov05]
- Goal:
  - Decouple shading from geometric complexity

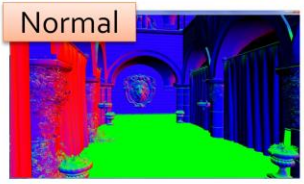
(Traditional) Deferred Shading

- Key idea
  - Defer shading computations
- First do a geometry pass
  - Sample geometry as per normal
  - But, no shading
  - Store geometry attributes per pixel (position, normal, albedo)
- Then do a shading pass
  - Use attributes from first pass.
  - Process lights one at a time.
    - Treat lights as geometry!

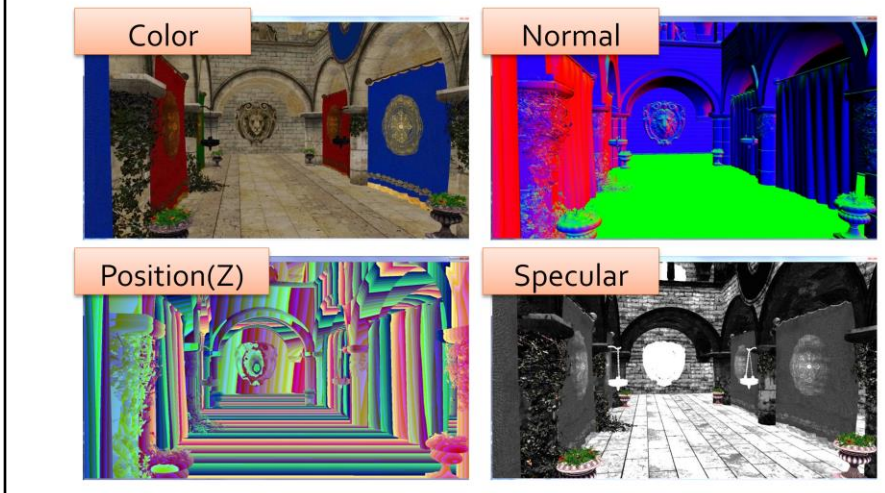
Color

Position(Z)

Normal



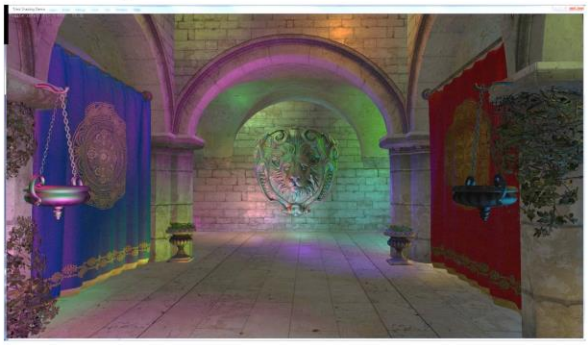
Render G-Buffers



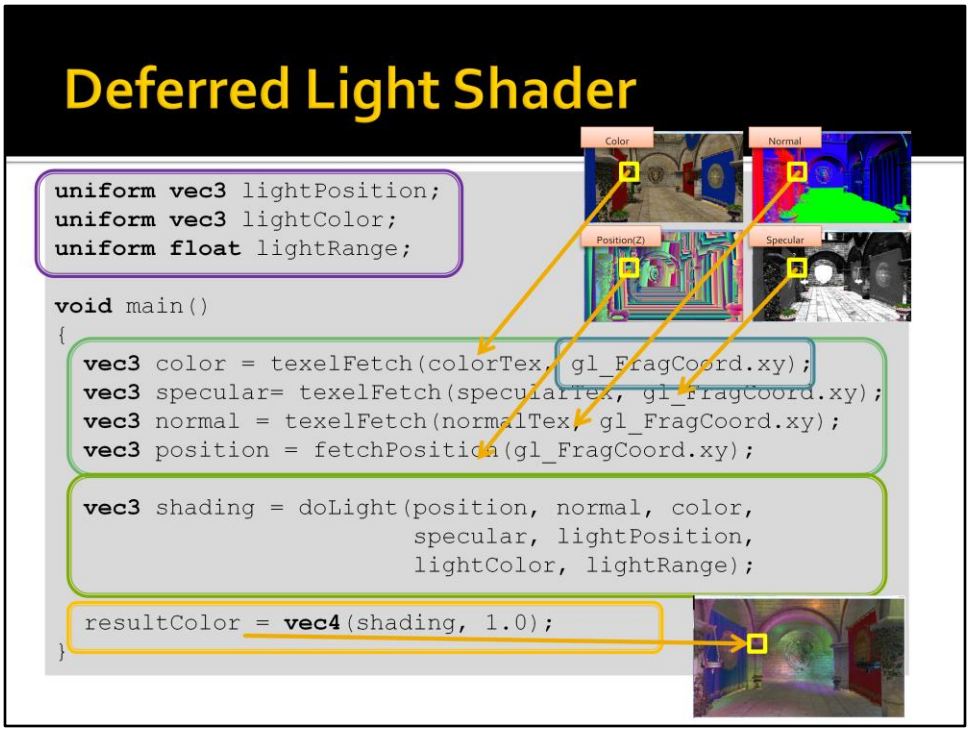
- So the geometry pass populates the G-Buffers with sampled geometry attributes.
- These are all the attributes that can change per pixel, and in addition a material ID can be used.

Draw Light Bounds

- For each light
  - Draw Light Bounds
    - For each fragment
      - Read G-Buffers
      - Compute Shading
      - Add to frame buffer



• Next we draw the bounding volumes of the lights, as polygon geometry to the screen.
•In the fragment shader of these we compute the shading.
•Which is then blended into the frame buffer.



- Here is a somewhat simplified light shader.
- So this is a fragment shader that is used when drawing a *light*.
- It is thusly executed once for each pixel covered by the light bounding sphere.
- The shader has as uniform parameters the light attributes, position etc, these are the same for each fragment.
- First all attributes are fetched from the G-Buffers.
    - Note that the screen space coordinate of the fragment is in `gl_FragCoord.xy`
- Then shading is computed using these and the uniform attributes of the light.
- And output to the color of the pixel.

Important Optimizations

- Stencil buffer opt [Arvoo3].
  - Analogous to stencil shadows.
  - Draw

Summary: Deferred Shading

THE GOOD

- Enables many lights
- Trivial light management
- Simple (light) shader management
- No overdraw
- Shadow map reuse
- Single geometry pass

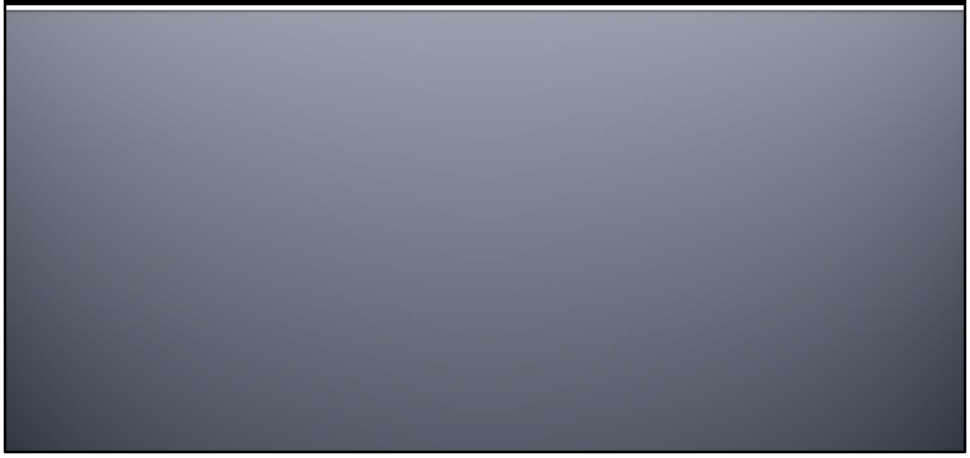
THE BAD

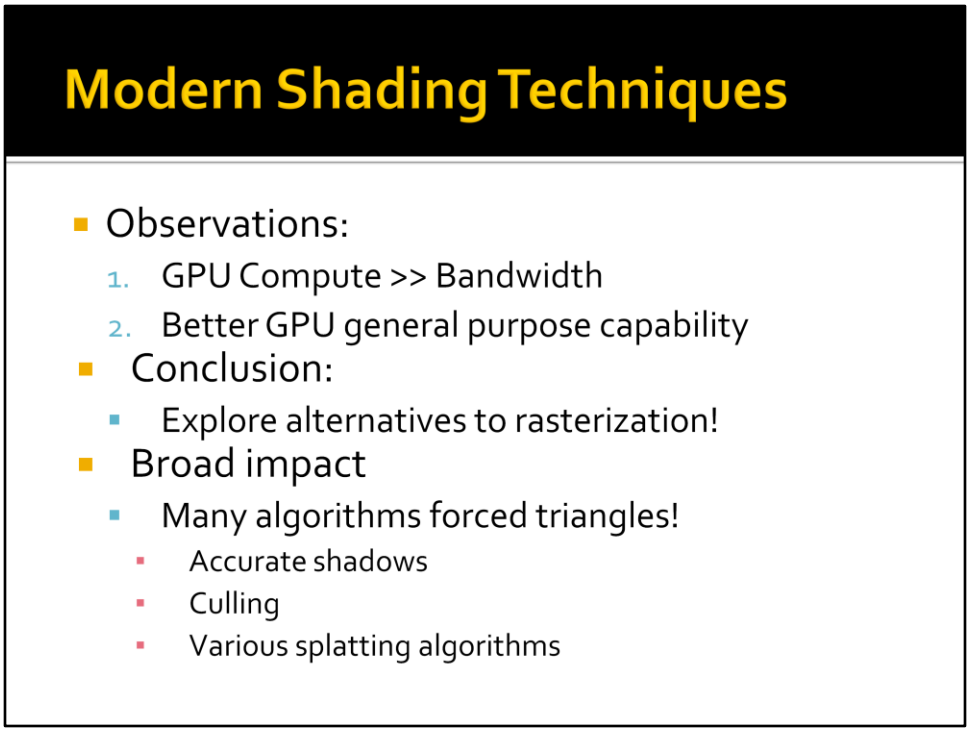
- High memory bandwidth usage
- No transparency
- Large frame buffer
  - Especially with MSAA
- Difficult to do multiple shading models
  - Custom shaders
- Accumulates light in frame buffer
  - High precision needed

Other Variants

- Deferred Lighting
  - Factor out specular and diffuse color
  - G-Buffer only store normal and shininess
  - Output diffuse and specular shading
  - Second geometry pass which multiplies colors
- Light PrePass
  - Much like the above
  - But with monochromatic specular highlight
- Similar performance as deferred
  - Only improves constant factors.
- Limits shading model even further

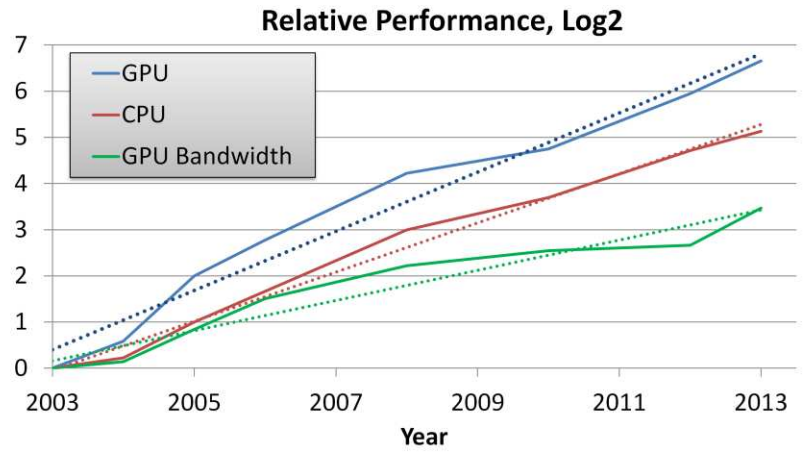
Modern Shading Techniques





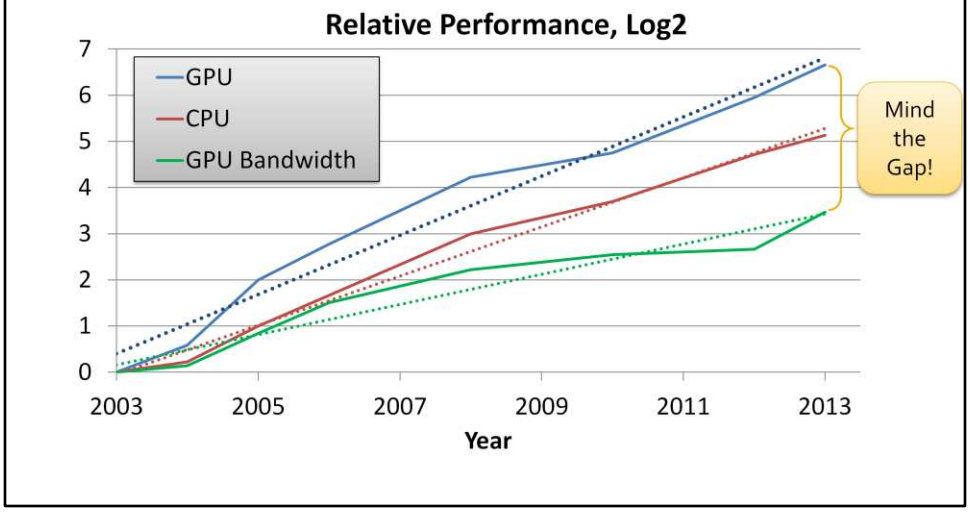
- The modern approaches to real-time shading with many lights all take their in the following 2 observations
    - First, GPU Compute Capacity is greater than the memory bandwidth, and grows faster .
    - Second, the GPU general purpose programming models and power of these cores has improved tremendously over the last years.
- This leads to the conclusion that we ought to explore more clever alternatives to rasterization based techniques, such as deferred shading, because they are bandwidth intensive.
- These observations have much broader impact as because of how GPUs were designed not long ago our community has spent a lot of effort developing algorithms that map well to triangle rasterization, even when this has been less than intuitive. To do so was simply the only path to real-time performance, as evidenced by the many examples, and I'm sure you can think of several of your own.
- Today this is not the case, with indirect drawing and programmable shader cores, we can, and must, revisit the same problems with a fresh approach.

GPU Performance Trends

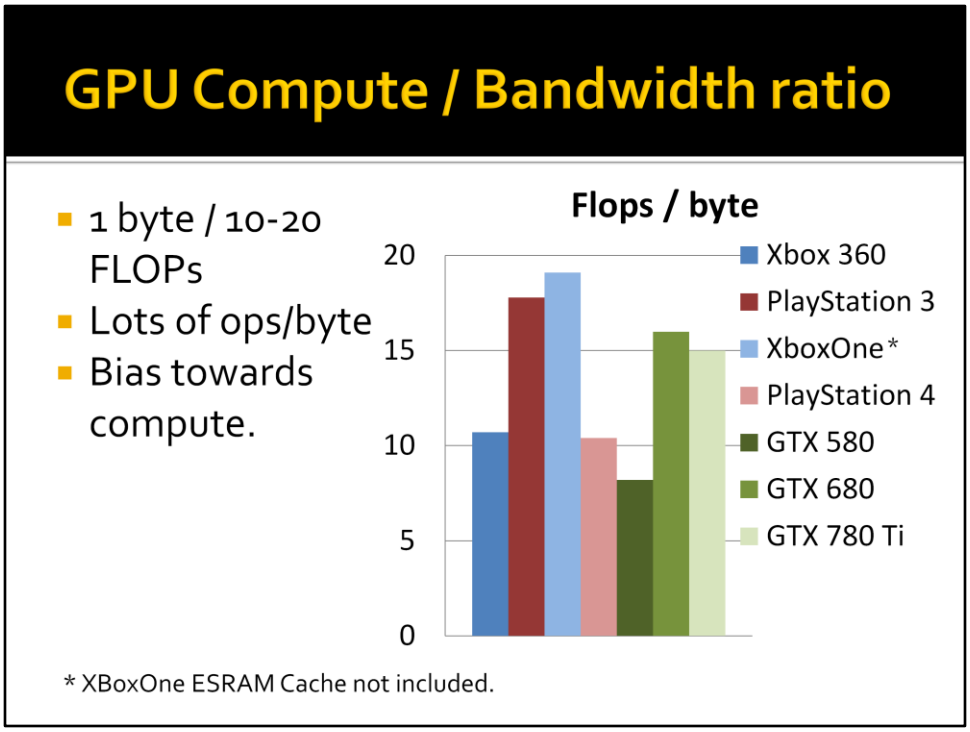


- To illustrate this process, this graph shows the relative performance trends of intel enthusiast level CPUs and NVIDIA GPUs.
- The green line also plots the memory bandwidth of GPUs,
- As this is a logarithmic plot, we see a fairly clear exponential growth in all three.

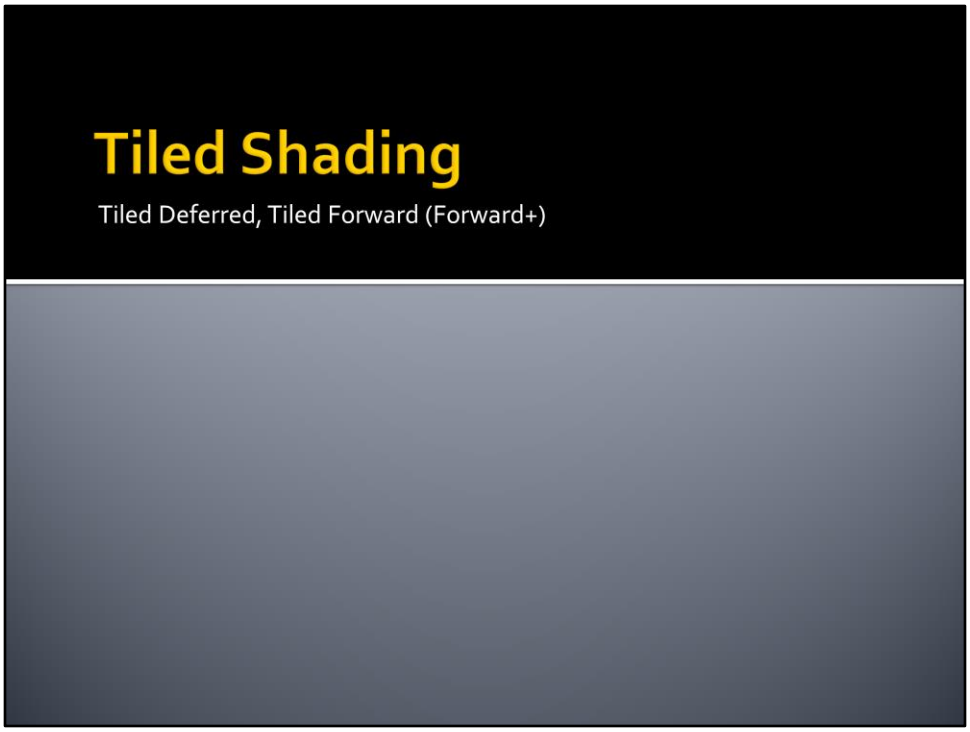
GPU Performance Trends



- However, note the great difference in exponent!
- This means that compute capacity is continually outpacing memory bandwidth,
- Any algorithm that is bottle necked by bandwidth will scale along this line.
- Whereas a compute bound algorithms will scale much, much, better.
- So we need to be mindful of this widening gap.



- Currently the gap is a span of some 10 to 20 floating point operations that can, or must be, performed for each byte of data loaded to reach peak performance.
- So if we load a single float, we need to do about 40 to 80 operations locally before fetching another…
- Of course texture caches and constant registers help a lot, so it is never quite that simple…
- …but the trend is clear and shows no sign of slacking off.



- This brings us to the first of the modern techniques that has been developed in this new, bandwidth constrained and compute oriented, landscape.
- Collectively called Tiled Shading, it covers both deferred and forward variants,
- The forward variant has unhelpfully been re-branded Forward+ by AMD, which obscures its nature.

The Bandwidth problem

- New type of overdraw
  - Light overdraw
- N lights cover a certain pixel ->
  - N reads from the **same** G-Buffer location
- Deferred Shading

```
for each light
  for each covered pixel
    read G-Buffer
    compute shading
    read + write frame buffer
```

- To motivate tiled shaing, we will look at why traditional deferred shading is bandwidth bound (or will be…)
- As we are now drawing the lights, in the shading pass, we have overdraw when many lights overlap the same pixel.
- Schematically deferred shading looks like this, we iterate over each light,
- and then in parallel, by drawing the bounding volume, over all the fragments.

The Bandwidth problem

- New type of overdraw
  - Light overdraw
- N lights cover a certain pixel ->
  - N reads from the **same** G-Buffer location
- Deferred Shading

```
for each light
  for each covered pixel
    read G-Buffer
    compute shading
    read + write frame buffer
```

- Lets examine traditional deferred shading to se why it is bandwidth bound (or will be…)
- As we are now drawing the lights, in the shading pass, we have overdraw when many lights overlap the same pixel.
- Schematically deferred shading looks like this, we iterate over each light,
- and then in parallel, by drawing the bounding volume, over all the fragments.

The Bandwidth problem

```
for each light
  for each covered pixel
    read G-Buffer
    compute shading
    read + write frame buffer
```

• The problem is from the fact that the innermost loop is over the pixels,
• This which requires repeated reading, and writing of the G-Buffers and frame buffer.
•So it is pretty clear that we need to get this out of the inner loop somehow, especially since G-buffers contain lots of data.

The Bandwidth problem

```
for each light
  for each covered pixel
    read G-Buffer
    compute shading
    read + write frame buffer
```

Re-order loops
Load/store -> Outer loop

```
for each pixel
  read G-Buffer
  for each affecting light
    compute shading
  write frame buffer
```

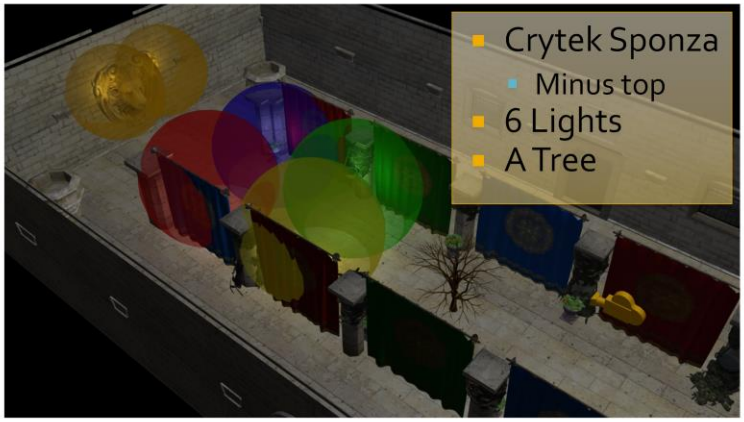
•So we want to re-arrange this loop to make the innermost loop iterate over the lights instead.
•Then we can hoist the G-Buffer read to the outer loop, only reading a single time.<click>
•Then we get this nice compute oriented loop.
•And finally a single write.
•This would effectively eliminate the bandwidth problem.
•So how do we go about this in practice?

The Bandwidth problem

- Requires
  - sequential access to lights for each pixel
- Global list
  - Inefficient
- List per pixel
  - Lots of data
  - Slow construction
- Share list in screen space tiles
  - E.g. 32 x 32 pixels
  - Simple construction
  - Little storage
    - 1 list / 1024 pixels
  - Coherent access

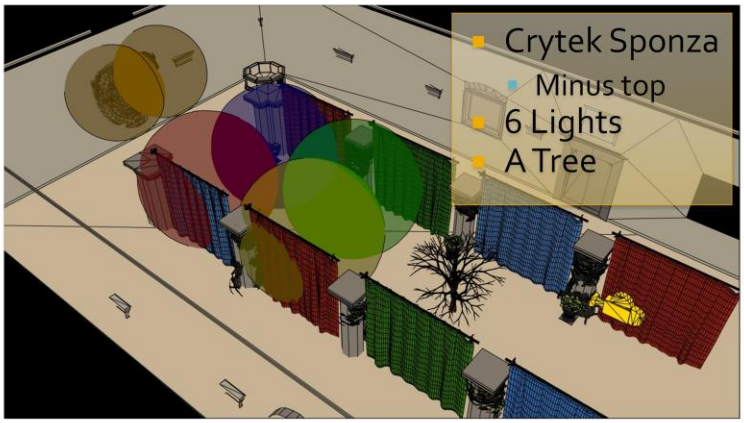
•We now need to access all the relevant lights for each pixel sequentially.
•Just using a global list of lights is of course terribly inefficient.
•At the other end of the spectrum, creating lists of lights for each pixel individually is both slow and requires lots of storage.
•Tiled shading strikes a balance, where we create lists for tiles of pixels.
•The list must be conservative, storing all lights that *may* affect any sample within the tile.
•So we trade some compute performance for bandwidth,
•which as we have seen is a pretty good gambit on modern GPUs.

Example Scene

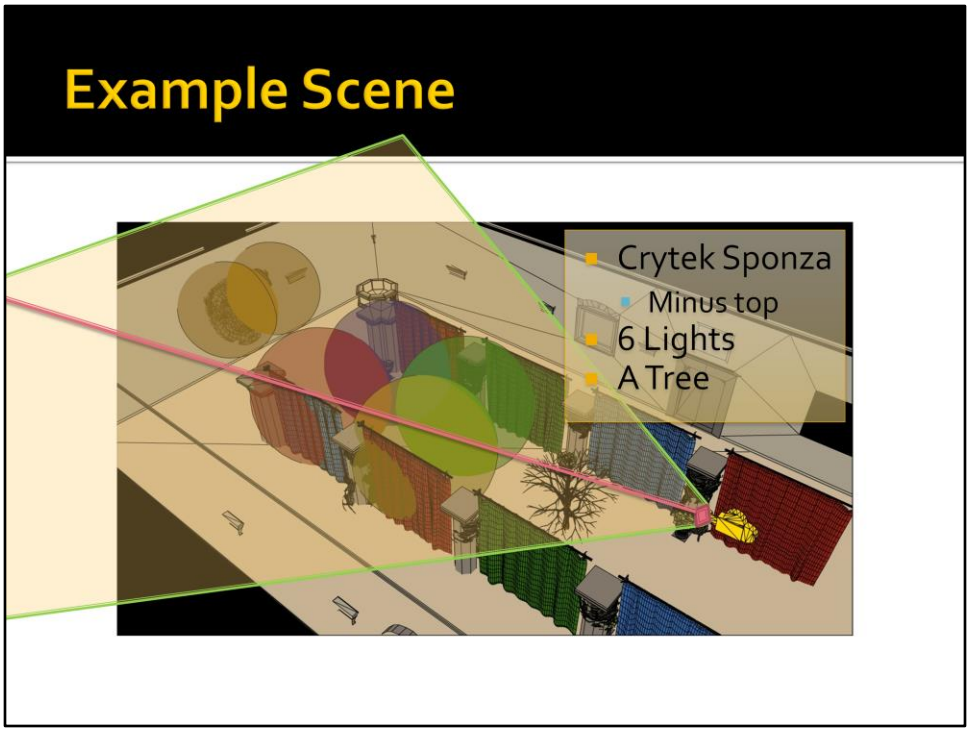


- We will be using this example scene to illustrate how tiled and clustered shading works.
- The scene is the usual Crytek Sponza scene, but with the top three quarters lifted off, to let us get a better view.
- I've added six lights of different colours, represented as spheres.
- To make the view more interesting, I've also added a tree.

Example Scene



- The same view in outline mode, to make it more clear.



This is the viewpoint that will be used to demonstrate the algorithms, the yellow camera is looking through the tree towards the lion head on the wall.

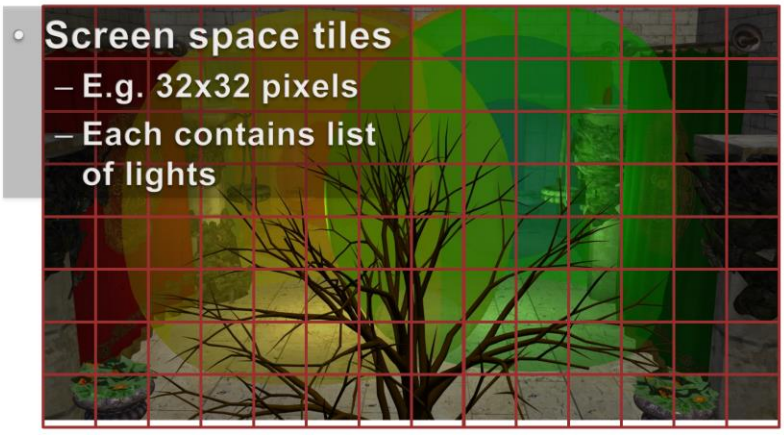
Example Scene

- Shifting to that point of view.
- We see that there is a tree near the camera and then we're looking through the 6 lights to the wall in the background.

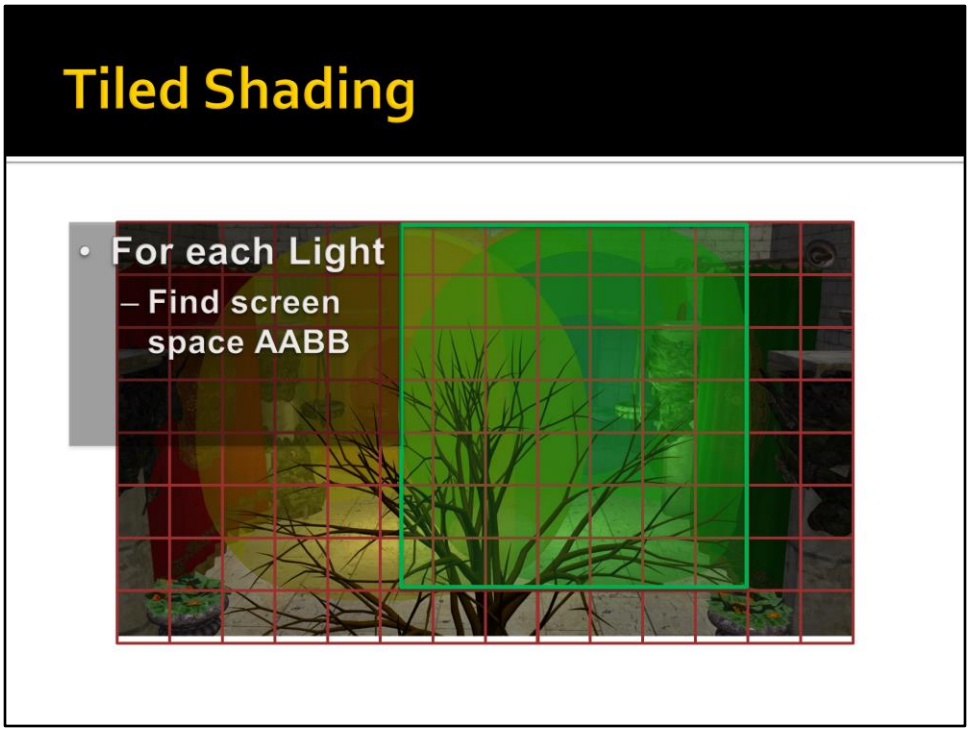
Tiled Shading

- With this I will summarize the tiled shading algorithm.
- Tiled shading can be implemented using either deferred of forward shading techniques, or a mix.
    - Note that AMD insists on calling "Tiled Forward Shading", "Forward+", it is however identical.
- The algorithm is conceptually very simple, and also quite easy to implement, at least in the simplest form.
- Performance can be very good, given the right circumstances.
- And fundamentally it is a 2D algorithm (we'll come back to this).

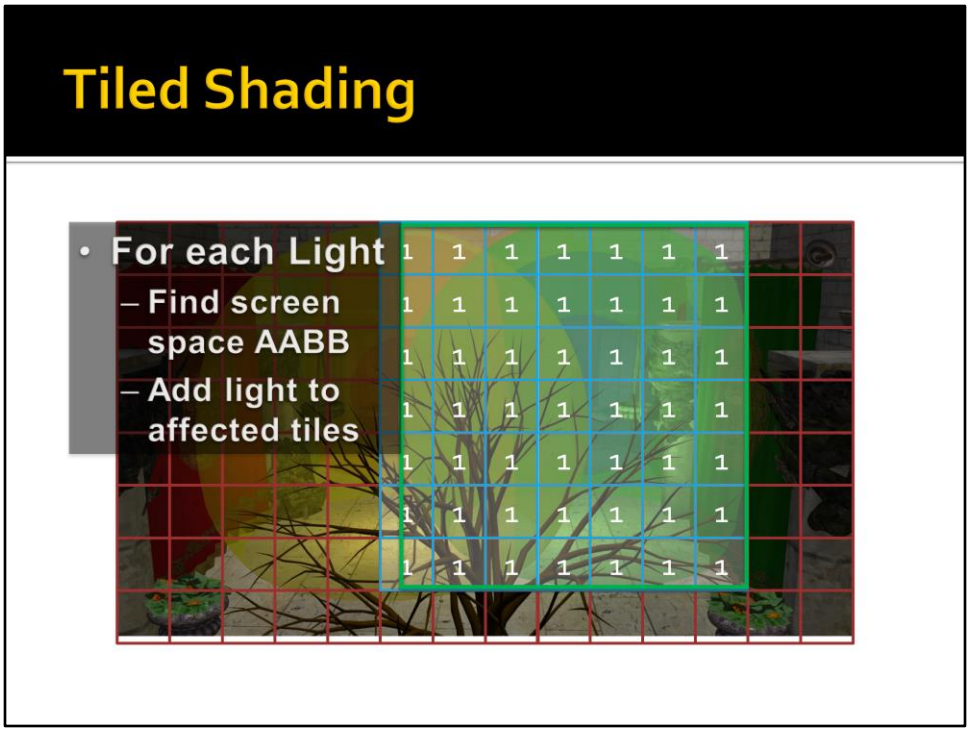
Tiled Shading



- The screen is divided into tiles, each covering say 32 by 32 pixels.
- Each tile contains a single list of all the lights that might influence any of the pixels inside.
- Note that this list is shared between the pixels, so overhead for list maintenance and fetching is low.

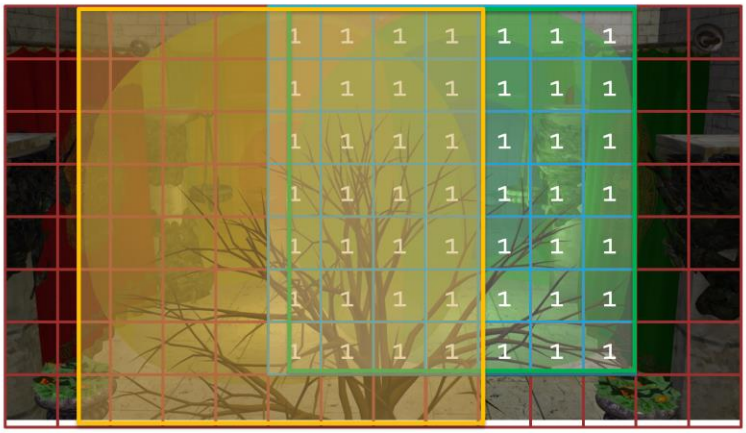


- To construct the lists, we might do as follows.
- For each light, establish the screen space bounding box, illustrated for the green light.



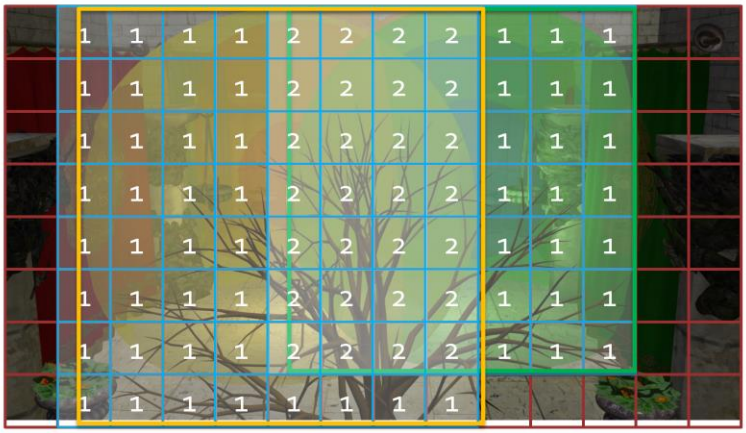
Then add the index of the light to all of the overlapped tiles.

Tiled Shading



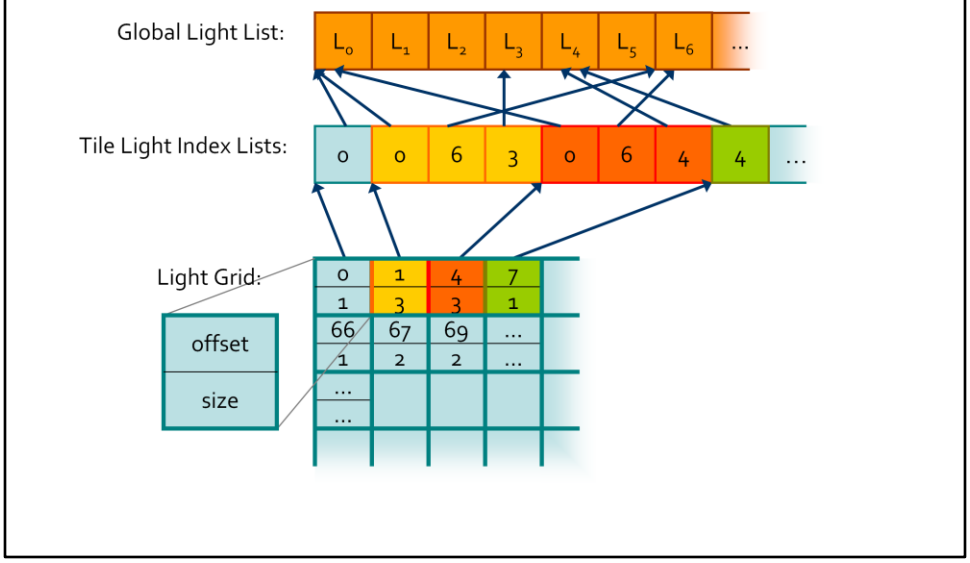
Then repeat this process for all remaining lights.

Tiled Shading



- The illustration only shows the counts, so you need to imagine the lists being built as well.
- In practice we'd also do a conservative per-tile min/max depth test, to cull away lights occupying empty space.

Tiled Shading Data



- After we have processed all the lights, we end up with a 2D grid such as this
- Each cell stores an offset and count that represent a range in a global buffer.
- This range contains a list of light indices indicating all the lights that the may affect the samples in the tile.

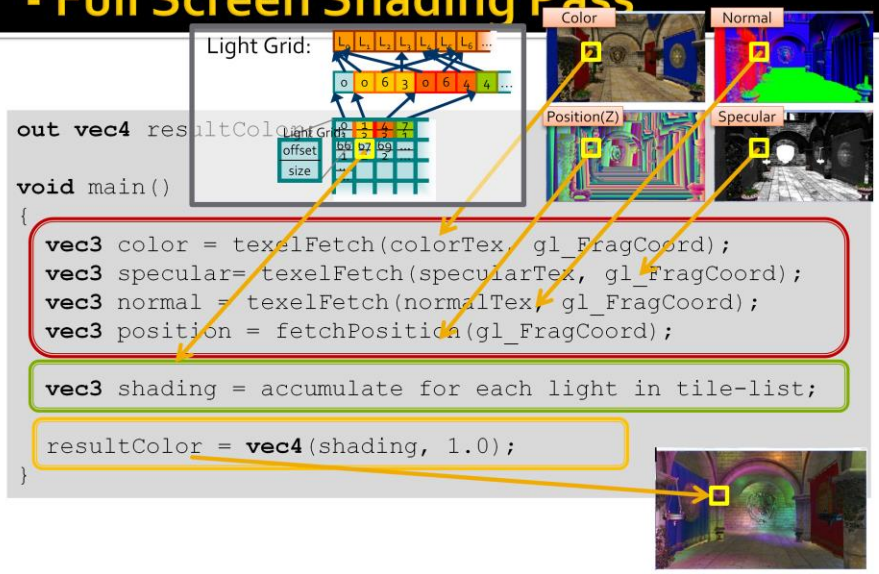
Tiled Shading

- Light Grid Provides
  - Access to light list for each pixel
  - List shared within tile
    - Low memory usage
    - Good access coherency
    - Not pixel exact
- Light Grid building
  - Is pretty quick
  - CPU for hundreds of lights

Tiled Deferred Shading

- 1. Render Scene to G-Buffers
  - Store geometry attributes per pixel
    - G-Buffers
- 2. Build Light Grid
- 3. Full Screen Quad (or CUDA, or Compute Shaders, or SPUs)
  - For each pixel
    - Fetch G-Buffer Data
    - Find Tile
    - Loop over lights and accumulate shading
    - Write shading

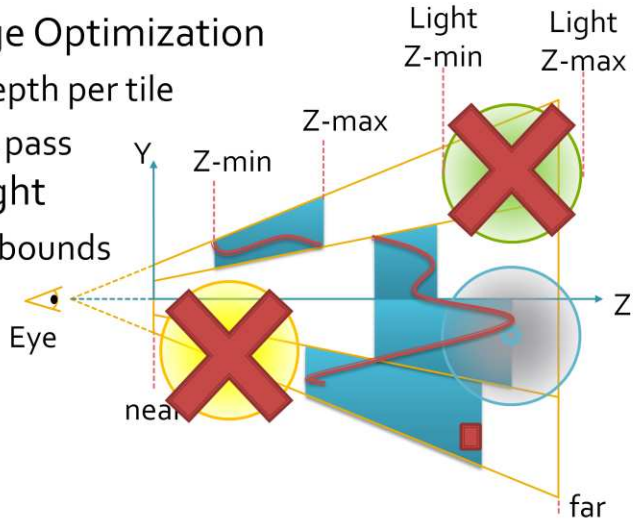
Tiled Deferred Shading
- Full Screen Shading Pass




```
out vec4 resultColor;

void main()
{
  vec3 color = texelFetch(colorTex, gl_FragCoord);
  vec3 specular= texelFetch(specularTex, gl_FragCoord);
  vec3 normal = texelFetch(normalTex, gl_FragCoord);
  vec3 position = fetchPosition(gl_FragCoord);

  vec3 shading = accumulate for each light in tile-list;

  resultColor = vec4(shading, 1.0);
}
```

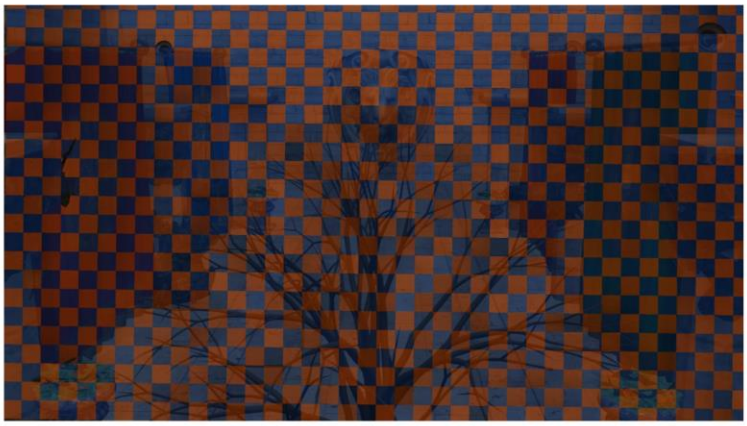
Tiled Deferred Shading -Depth Range Optimization

- Depth Range Optimization
  - Min/Max depth per tile
  - From Pre-z pass
- For each Light
  - Test depth bounds



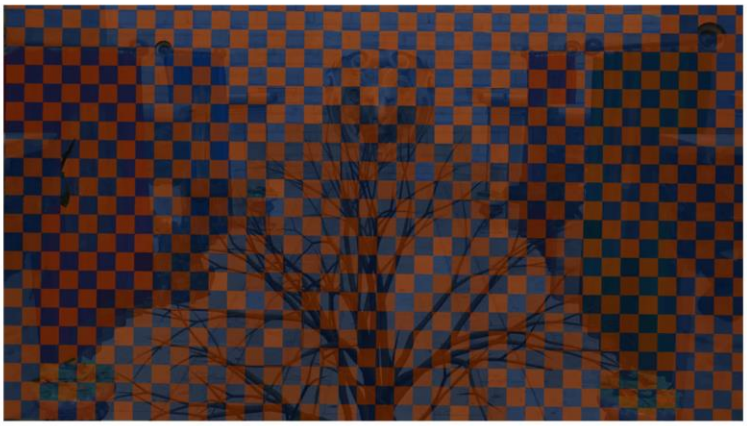
- Tiles in 1D, from side
- View Frustum
- 4 subdivisions
- Redline is geometry
- Min and max depth per tile
- Light range, rejected, completely hidden
- Another rejected, completely in front
- Rejected in one tile, not others

Tiled Shading Problems



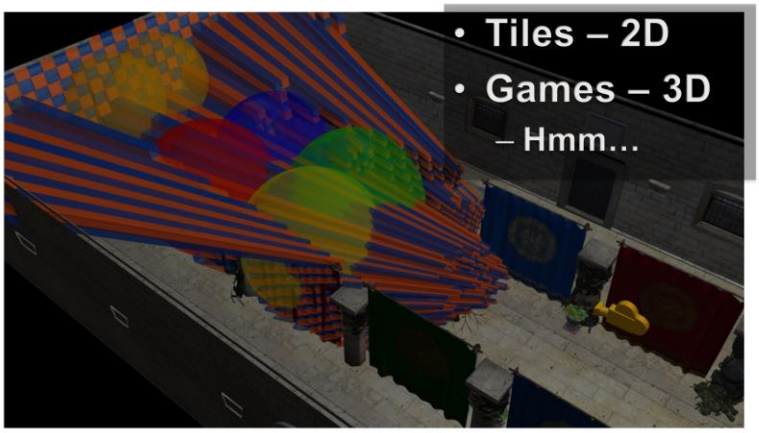
- This is a view of the tile geometry pulled out of our implementation.

Tiled Shading



- Switching to the overhead view, we see how they extend from the tree over to the background geometry.

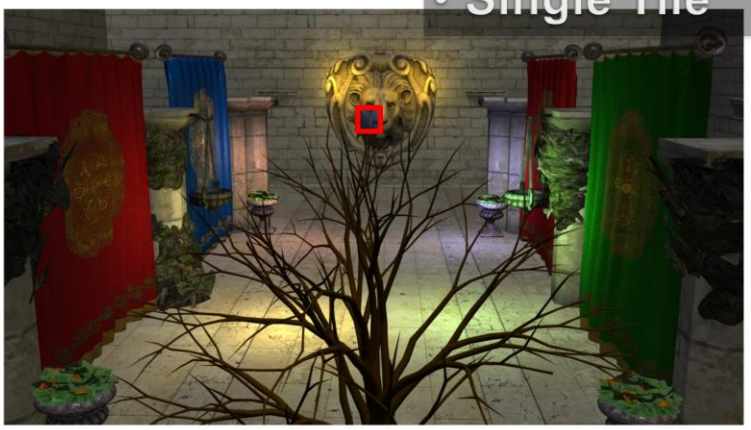
Tiled Shading Problems



- Toggling on the light geometry, we see that there is a lot of overlap…
- …even in the empty space behind the tree.
- We now should be able to start seeing the shape of the problem with 2D tiles in a 3D world.

Tiled Shading

- Single Tile



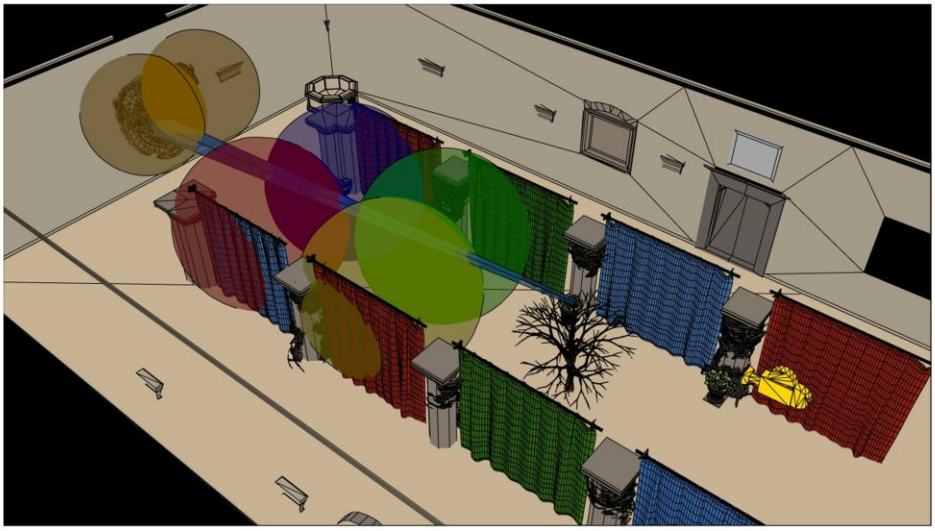
We'll now take a closer look at a single tile, in order to highlight the problem.

Tiled Shading

Tiled Shading

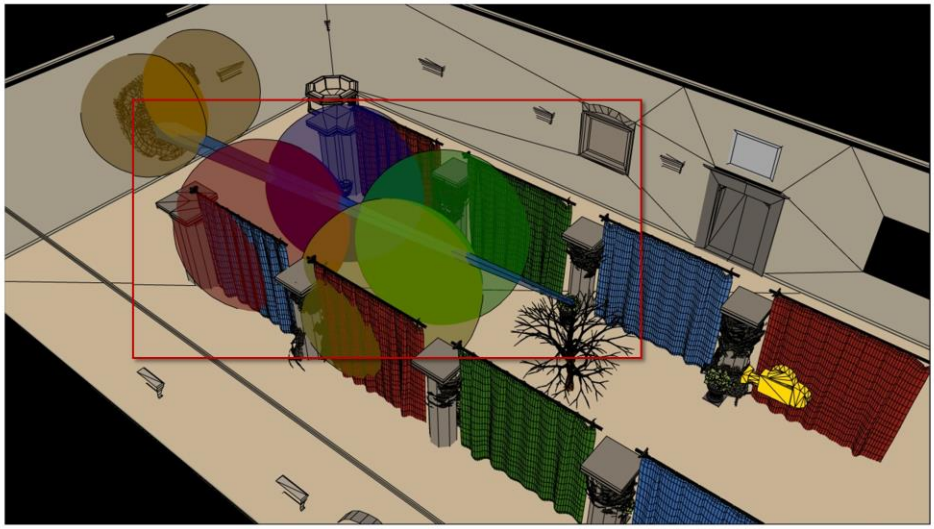
- As we can see, there is a small number of samples from the tree
- And the rest, the lion share of the pixels are in the background.

Tiled Shading
-The Discontinuity Dysfunction

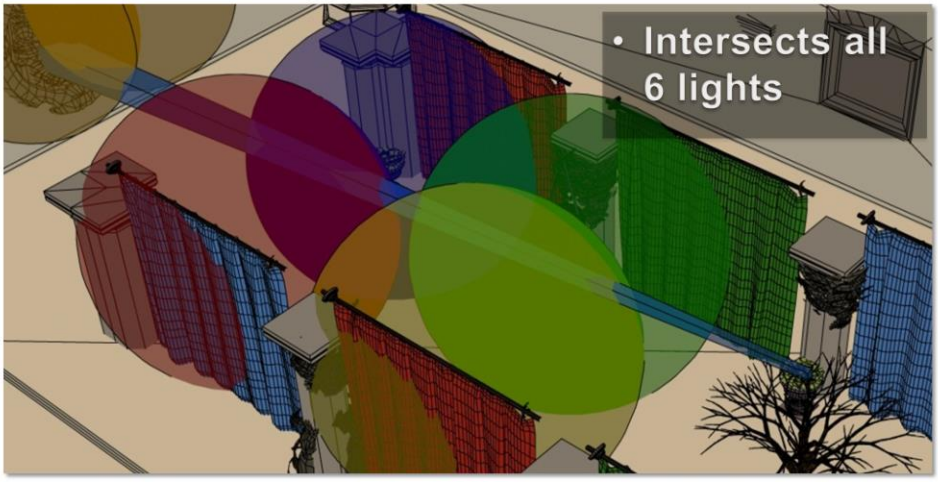


* In 3D the tile looks like this…

Tiled Shading
-The Discontinuity Dysfunction

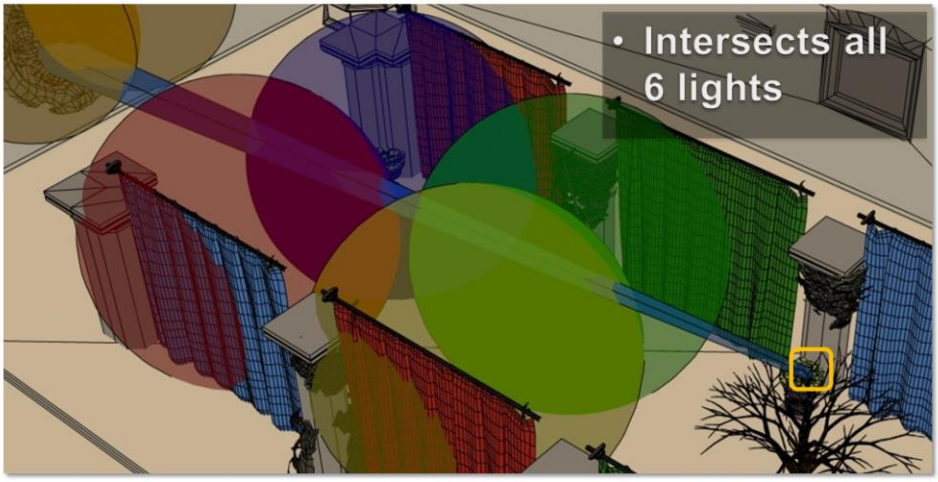


Tiled Shading
-The Discontinuity Dysfunction

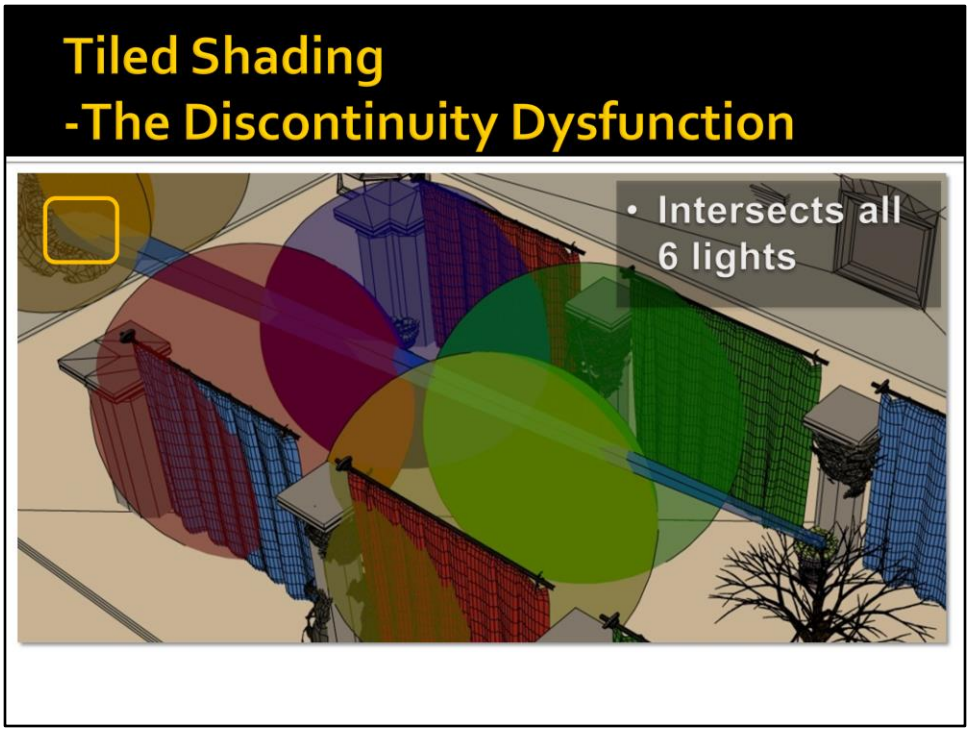


- In 3D the tile looks like this…
- Extending from the tree through the visible scene to the wall.
- This one troublesome tile intersects all 6 lights in our simple test scene.

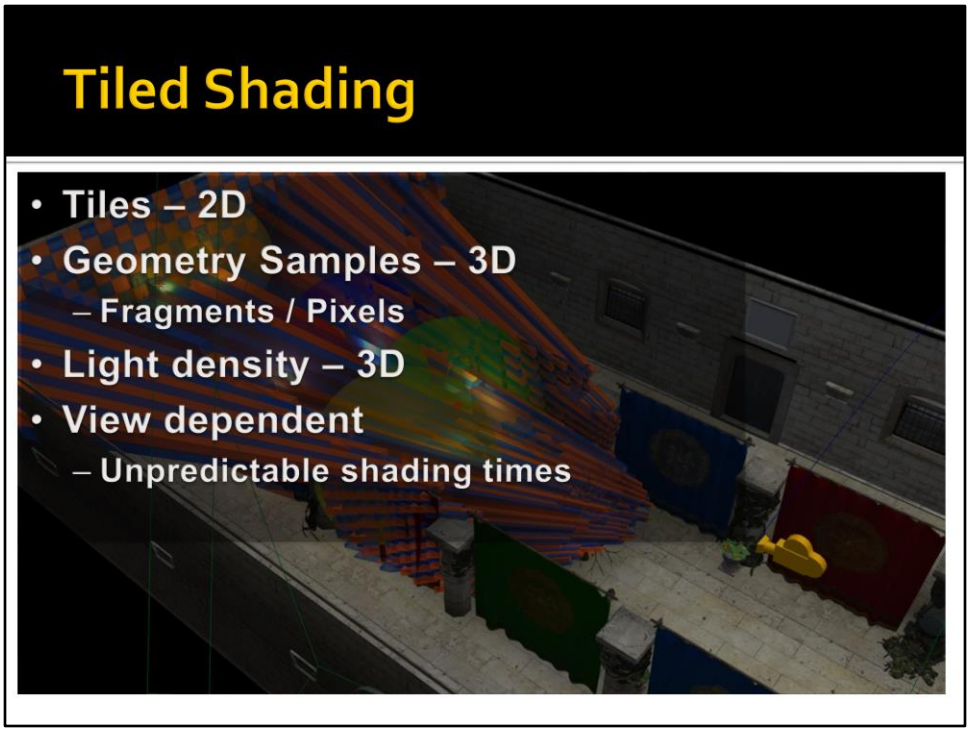
Tiled Shading
-The Discontinuity Dysfunction



- While actually some of the samples, from the tree, are affected zero of these lights.



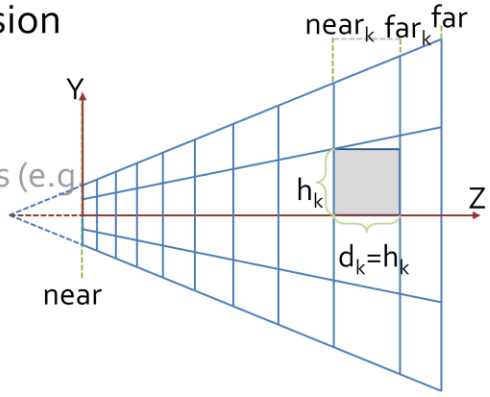
- And the rest, would only need two of the lights.



- So as we have shown, there is a fairly fundamental problem with tiled shading.
- The basic problem stems from that we are making the intersection between lights and geometry samples, both of which are 3D entities, in a 2D screen space.
- The main practical issue with this is that the resulting light assignment is highly view dependent. This means that we cannot author scenes with any strong guarantee on performance, as a given view of the scene may have a significantly higher *screen space* light density than average.
- For example, we'd like to be able to construct a scene with, say, maximum 4 lights affecting any part of the scene. In this case, we would like shading cost to be proportional to this, and stable, given different view points. Unfortunately, no such correlation exists for tiled shading.
- In other words shading times are unpredictable, which is a major problem for a real time application.

Clustered Shading – Key Idea

- Add the 3rd dimension
  - Tile also in depth direction = cluster
  - Also > 3 dimensions (e.g. normals)

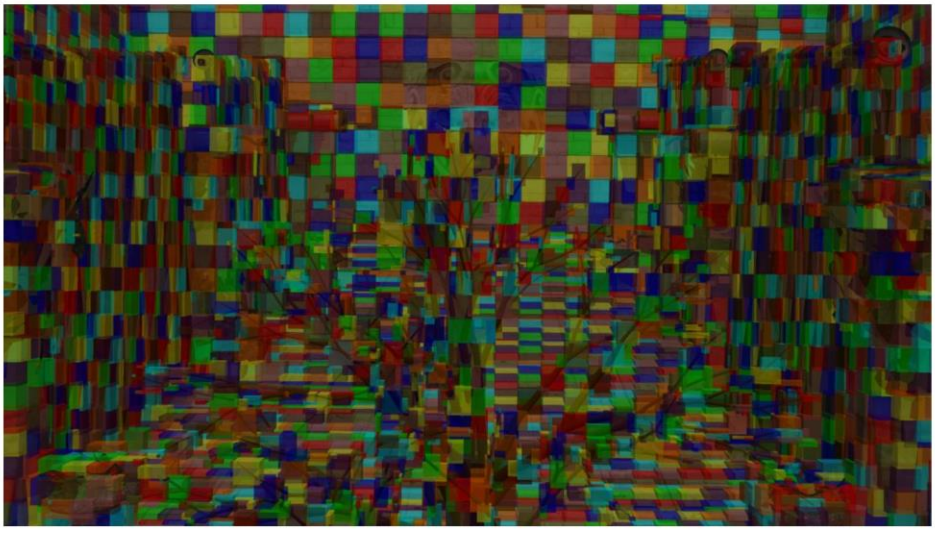


- So how do we solve this?
- This is the basic question for our clustered shading paper.
- A fairly obvious solution, given what has been said so far, is to use 3D tiles of some sort.
- It is less obvious what particular kind of subdivisions to use, and whether this will actually improve efficiency.
- Another question we explore in the paper is to tile in yet higher dimensions, based on normal direction as well.

Clustered Shading
The Solution

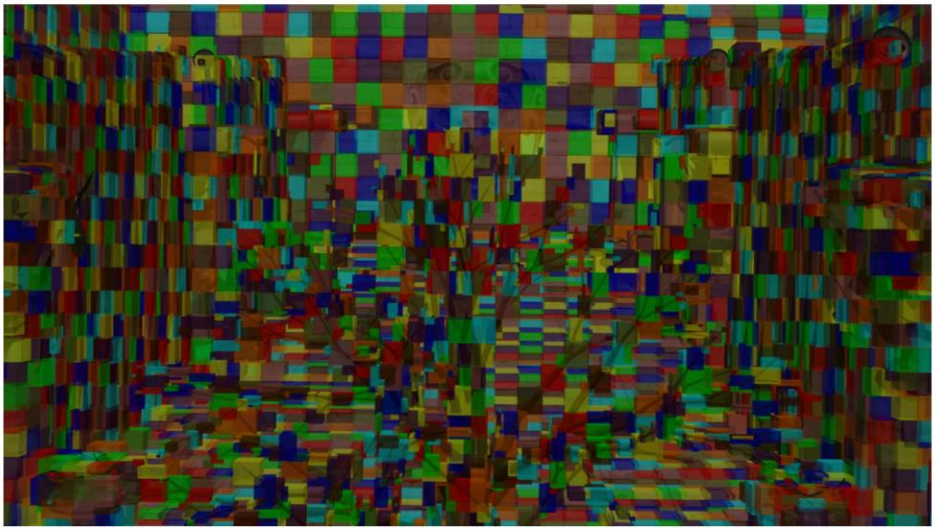
Compare the tiles shown here…

Clustered Shading
The Solution



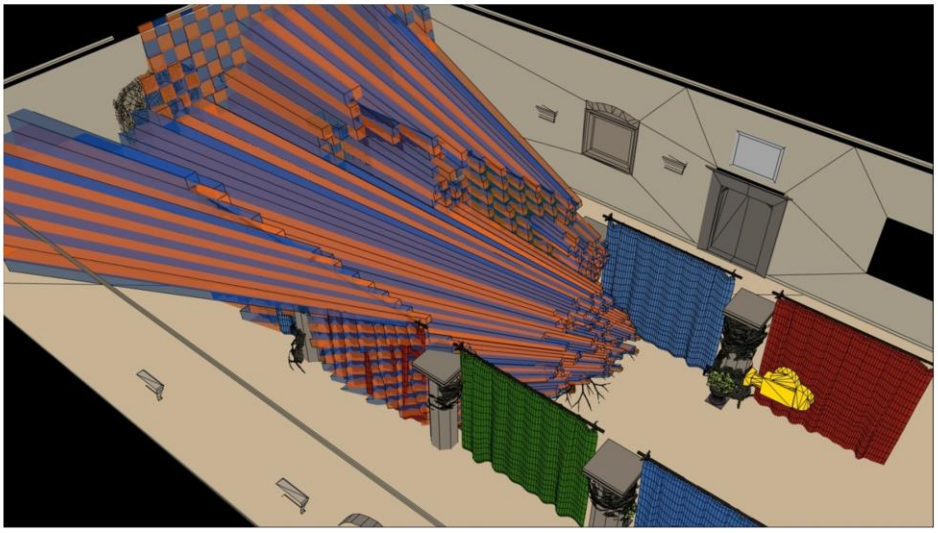
- …with the view space cluster AABBs shown here.
- Along the top edge of the screen, it is easy to see that the clusters and tiles are very similar, where the depth within each tile is shallow.
- In the middle, where the tree is, things are more interesting, as several layers of depth are visible.

Clustered Shading
The Solution



Going to the overhead view again.

Clustered Shading
The Solution

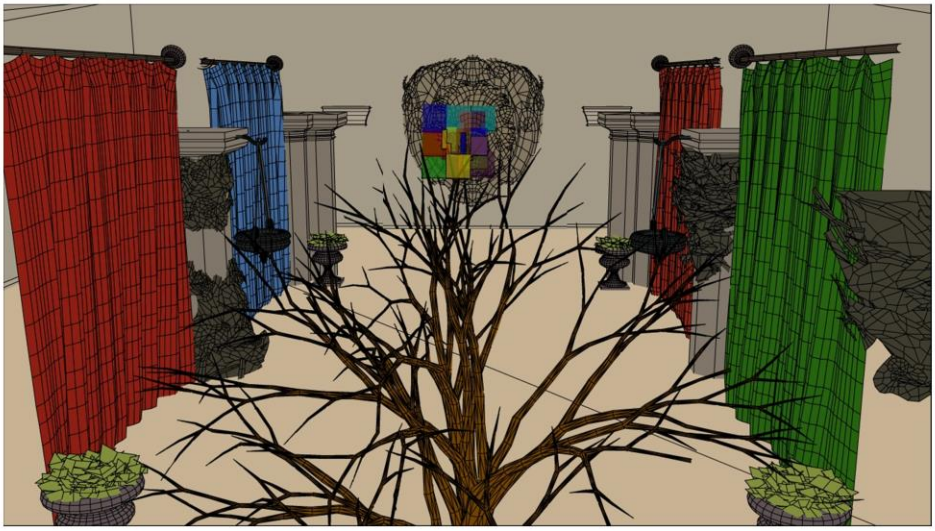


Comparing the tiles…

Clustered Shading
The Solution

- …to the clusters, shows that the clusters approximate the visible geometry a lot better.
- This means that the intersection with the light volumes ought to also be more precise.

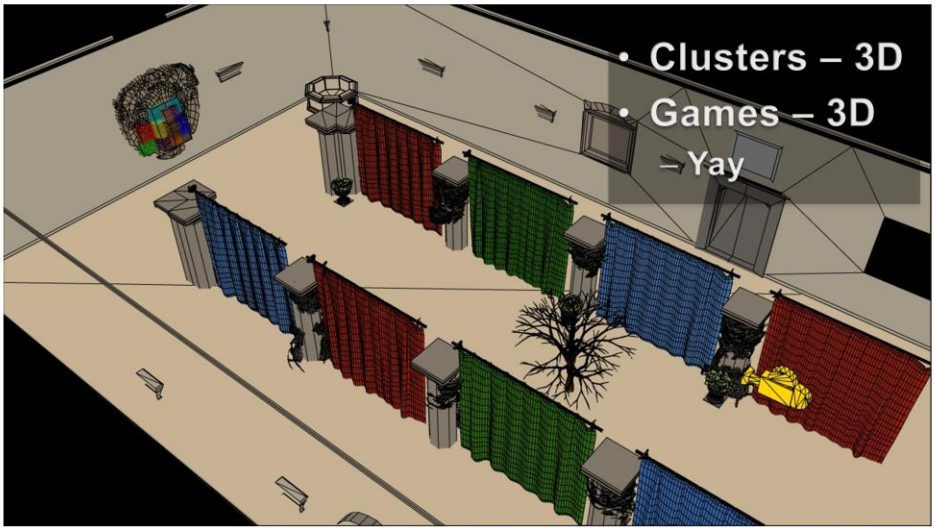
Clustered Shading
The Solution



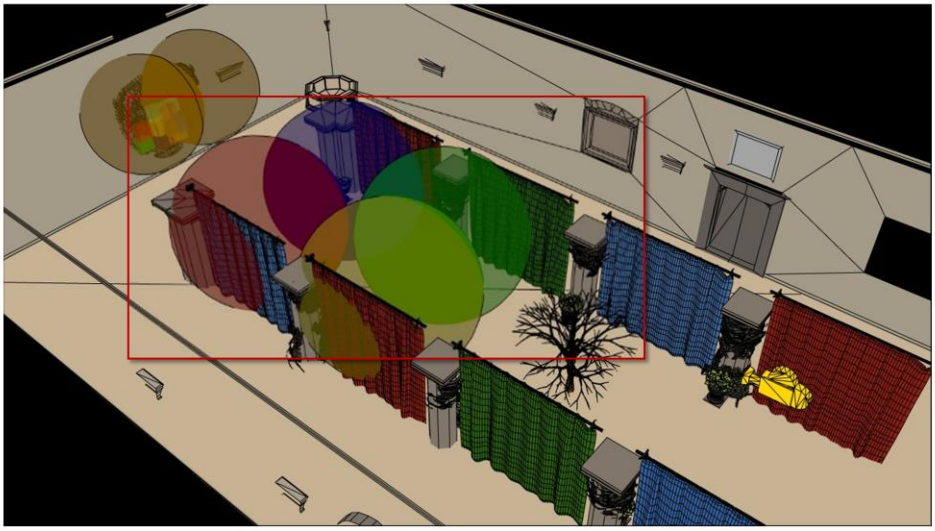
Going back to the single tile example, but with clusters.

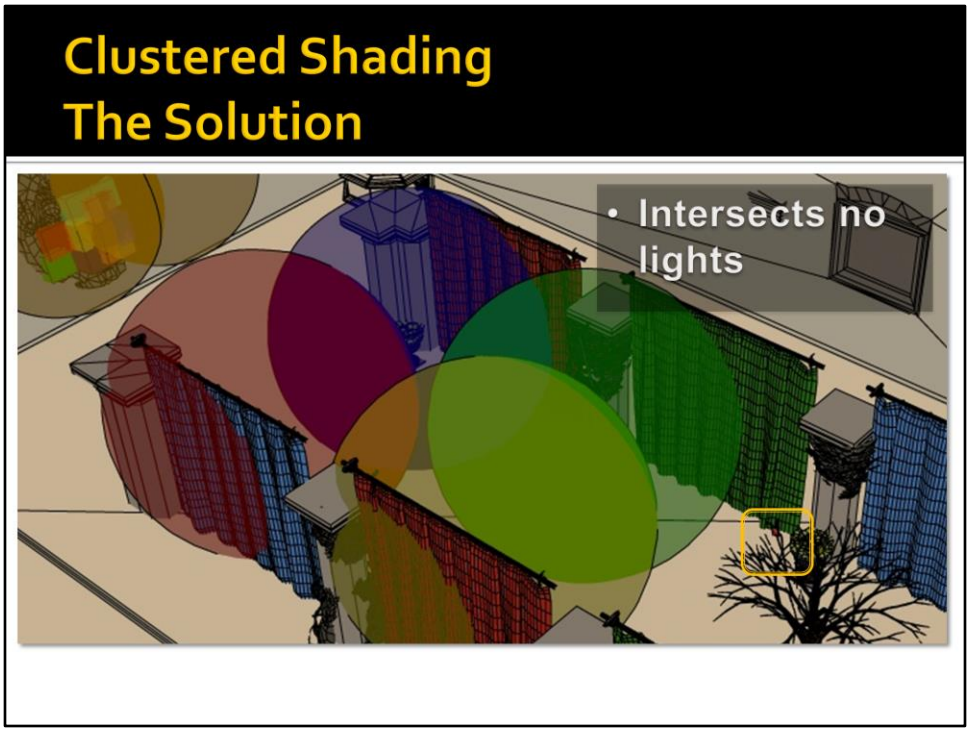
Clustered Shading
The Solution

Clustered Shading
The Solution

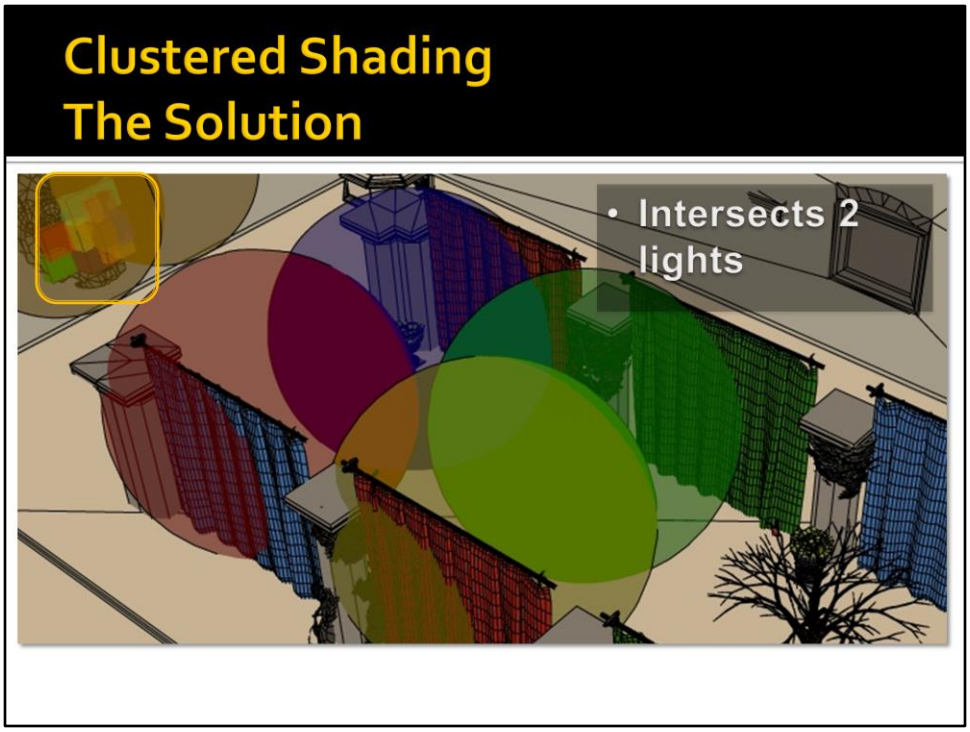


Clustered Shading
The Solution





We see that none of the lights overlap the clusters that are on the tree in the fore ground.



- And in the background, only the volumes of the two required lights intersect.
- Clearly clusters has a good potential for improving efficiency, we'll now need to talk about how they can be implemented.

Clustered Shading -Idea

- Add the 3rd dimension
  - Tile also in depth direction = cluster
  - Also > 3 dimensions (e.g. normals)
- Bounded volume around samples
  - Shading cost ~ Light density.
- New Challenges
  - Many more (potential) clusters
    - Must find those actually used
  - Adding lights no longer screen space

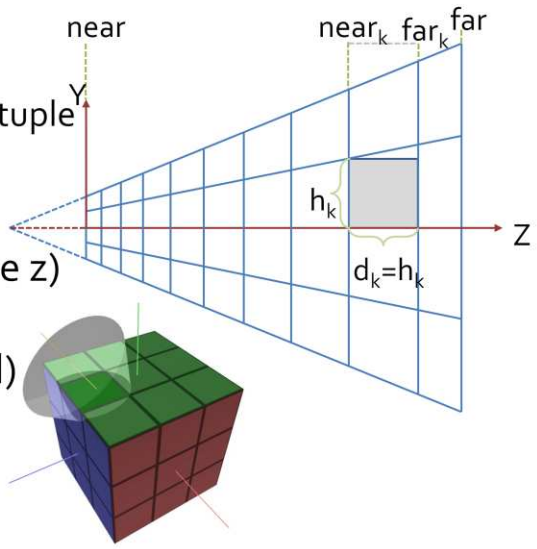
Clustered Shading – Algorithm

- Rasterize G-Buffers
  - Forward: pre-z pass
- Cluster assignment
- Find unique clusters
- Assign lights to clusters
- Shade view samples
  - Deferred: Full screen pass
  - Forward: Geometry pass

- This is a high level version of the clustered shading algorithm
- I've grayed out parts that are identical to tiled shading.
- Cluster assignment means to identify what cluster each sample belongs to, this is a simple mapping.
- Then we need to work out which of the clusters are represented by these samples, and
- Finally we assign lights to these clusters, ensuring we are not wasting storage and work assigning lights to empty clusters.
- Note that Emil will be presenting a rather different approach to implementing the algorithm, which leaves out and replaces the first two steps.

Cluster Assignment

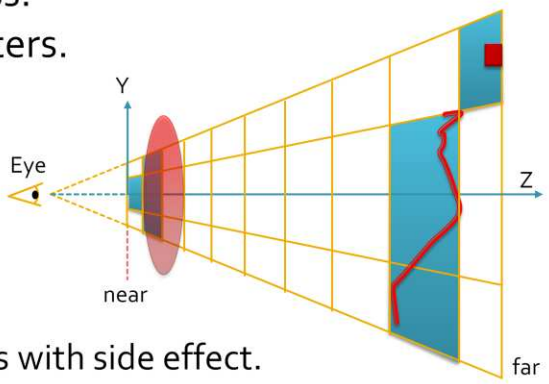
- Cluster key:
  - *ck* = (*i*, *j*, *k*) integer tuple
- *i*, *j* = 2D tile id
  - gl_FragCoord.xy
- *k* = ≈log(view space z)
- (quantized normal)



- Cluster assignments is a simple mapping from sample coordinate, to an integer tuple i,j,k
- i and j are simply the tile coordinates, which can be derived by dividing gl_FragCoord.xy by the tile size.
- k is a logarithmic function of the view space Z of the sample, not simply the logarithm, for the exact equation see the paper.
- We use this subdivision as it creates self similar clusters that are as cube like as possible. This makes them better suited for culling. The logarithmic subdivisions also means that as clusters become larger further away, we get a kind of LOD behaviour and do not end up with insane numbers of clusters, for a wide range of view parameters.

Finding Unique Clusters

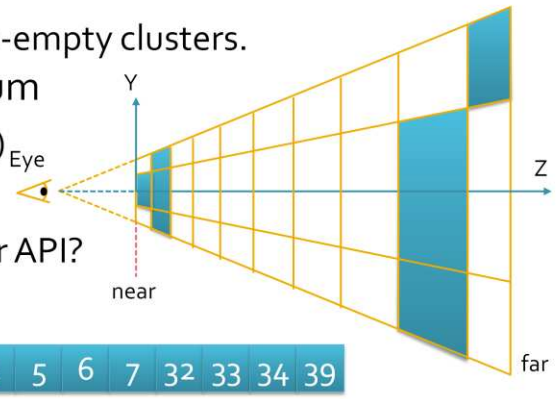
- Full screen pass.
- Flag used clusters.
  - Read depth
  - Compute *ck*
  - Set cell in grid to one.
- Forward:
  - Geometry pass with side effect.



- Finding the unique clusters is a full screen pass, which simply constructs the cluster key for each sample.
- And sets the corresponding cell in a 3D grid to 1. This grid is non-regular subdivision of the view volume.

Finding Unique Clusters

- Compact Non-Zero
  - Yields list of non-empty clusters.
- Parallel prefix sum
  - chag::pp (CUDA)
  - thrust (CUDA)
  - Compute Shader API?



- We then compact the grid into the list of non-zero elements.
- This can be implemented through a parallel perfix sum.
- Which is a very quick process, for the million elements or so needed.
  - Taking a fraction of a millisecond.
- This leaves us with a list of clusters which needs lights assigned to them.

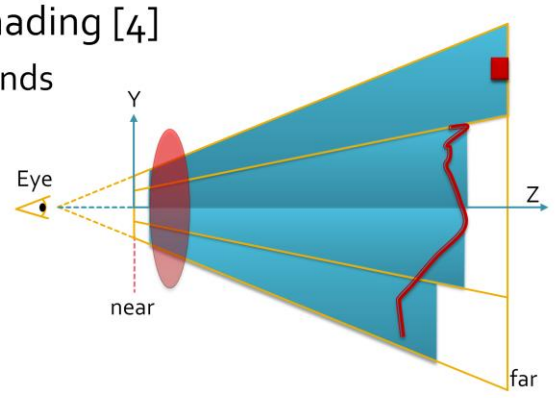
Light Assignment

- Many clusters
- Many lights
- Hierarchical approach
  - Hierarchy over lights
    - 32-way tree (matches SIMD of GPU)
    - Dynamically rebuilt on GPU
  - Each cluster traverses light tree
    - BV tests

- When pushing the limit on number of lights, a hierarchy over lights makes sense.
- We use a BVH with a branching factor of 32, which is rebuilt each frame.
- When not so many lights are used, there are many other approaches which may be better. Again Emil will show a rather different approach later on.

Transparent Geometry

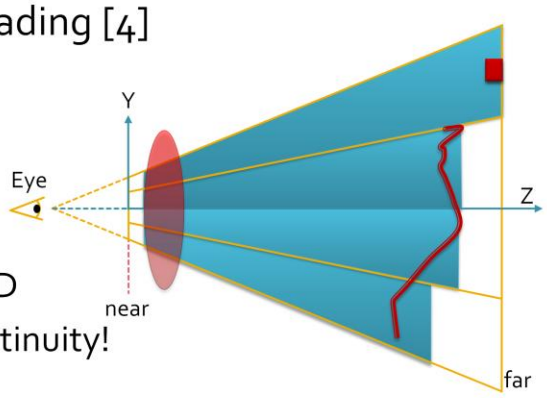
- Tiled Forward Shading [4]
  - Extend tile Z bounds
    - To Min Z
    - Or Near plane



- It's simple to support transparent geometry with tiled forward shading.
  - For example by finding the min and max Z values, or just extending tiles to the near plane.
  - However…

Transparent Geometry

- Tiled Forward Shading [4]
- Problems
  - View dependent
  - Degenerates to 2D
  - Full screen discontinuity!

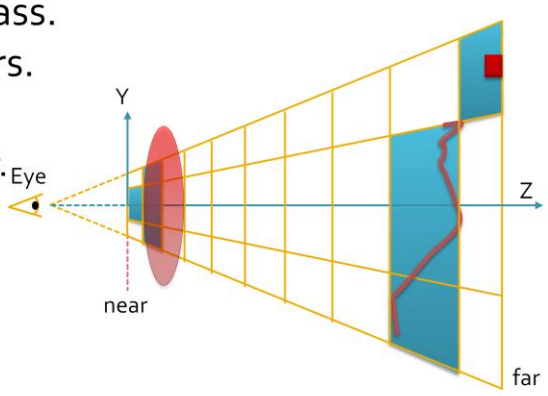


- The result of transparent geometry covering the view is effectively the loss of the depth range optimization
  - which can be a very bad thing™.

- Again, the biggest issue here is that it is view dependent, and so will only be possible to determine at run time
  - and, as we all know, this kind of problem usually starts showing up five minutes after we shipped a build to the publisher.

Clustered Forward Shading [4]

- Pre-geometry pass.
- Flag used clusters.
  - Side effect in fragment shader.
  - Set cell in grid to one.



- Clustered Shading does not suffer from this problem, as each cluster represents a fixed section of 3D space.
- The only problem we face is trying to determine what clusters are need such that we can assign lights to them.
- This can be done by rendering a pre-pass with the transparent geometry.
- This can be performed after a regular G-buffer or pre-z pass, with color and depth writes turned off, and sets the used clusters to one as a side effect in the fragment shader.
- The algorithm is otherwise as before.
- Note that clustered shading provides an efficient way to solve the light assignment problem for any transparency technique, not just good old order dependent transparency.

Clustered Shading - Summary

- High performance
- Low view dependence
- Good worst-case performance
- Fully Dynamic
- Supports transparency
- Forward / Deferred, or both
- No Pre-Z required

Technique Comparison

	Trad. Deferred	Tiled Deferred	Tiled Forward	Clustered Deferred	Clustered Forward
Bandwidth Use	High	Low	Low	Low	Low
Transparency	No	Maybe	Yes	Maybe	Yes
MSAA	Maybe	Maybe	Yes	Maybe	Yes
Shadow Map reuse	Yes	No	No	No	No
Geometry Passes	1	1	1–2*	1	1–2+
Register Pressure	Low	High	High	High	High
Innermost loop	Pixels	Lights	Lights	Lights	Lights
FB Precision	High	Low	Low	Low	Low
View dependence	Low	High	High	Low	Low
Vary Shading Models	Hard	Hard	Simple	Hard	Simple

References

[1] *Practical Clustered Deferred and Forward Shading*, Persson & Olsson 2013, http://advances.realtimerendering.com/s2013

[3] *Clustered Deferred and Forward Shading*, Olsson et.al. 2012

[4] *Tiled and Clustered Forward Shading*, Olsson et.al. 2012

[Valient14] *Reflections and volumetrics of Killzone Shadow Fall* , SIGGRAPH'14, http://advances.realtimerendering.com/s2014

[Andersson14] *Rendering Battlefield 4 with Mantle*, GDC 2014

[Shulz14] *Moving to the Next Generation - The Rendering Technology of Ryse*, GDC 2014.

[Tebbs92] Brice Tebbs, Ulrich Neumann, John Eyles, Greg Turk, and David Ellsworth. *Parallel architectures and algorithms for real-time synthesis of high quality images using deferred shading.* 1992.

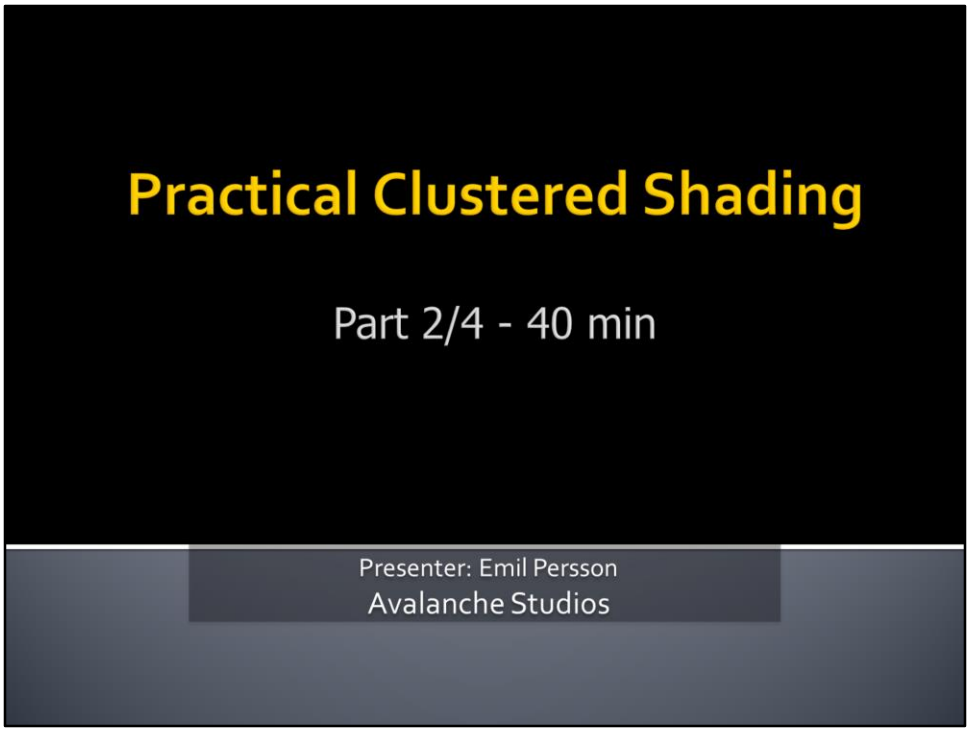
[Saito90] Takafumi Saito and Tokiichiro Takahashi. *Comprehensible rendering of 3-d shapes.* SIGGRAPH Comput. Graph., 24(4):197–206, 1990.

[Hargreaves04] Shawn Hargreaves and Mark Harris. *Deferred shading.* NVIDIA Developer Conference: 6800 Leagues Under the Sea, 2004.

[Shishkovtsov05] Oles Shishkovtsov. Deferred shading in S.T.A.L.K.E.R. In GPU Gems 2. Addison-Wesley, 2005.

[Arvo03] Jukka Arvo and Timo Aila. Optimized shadow mapping using the stencil buffer. journal of graphics, gpu, and game tools, 8(3):23–32, 2003.

Papers / Slides / Demo implementation with source: http://www.cse.chalmers.se/~olaolss



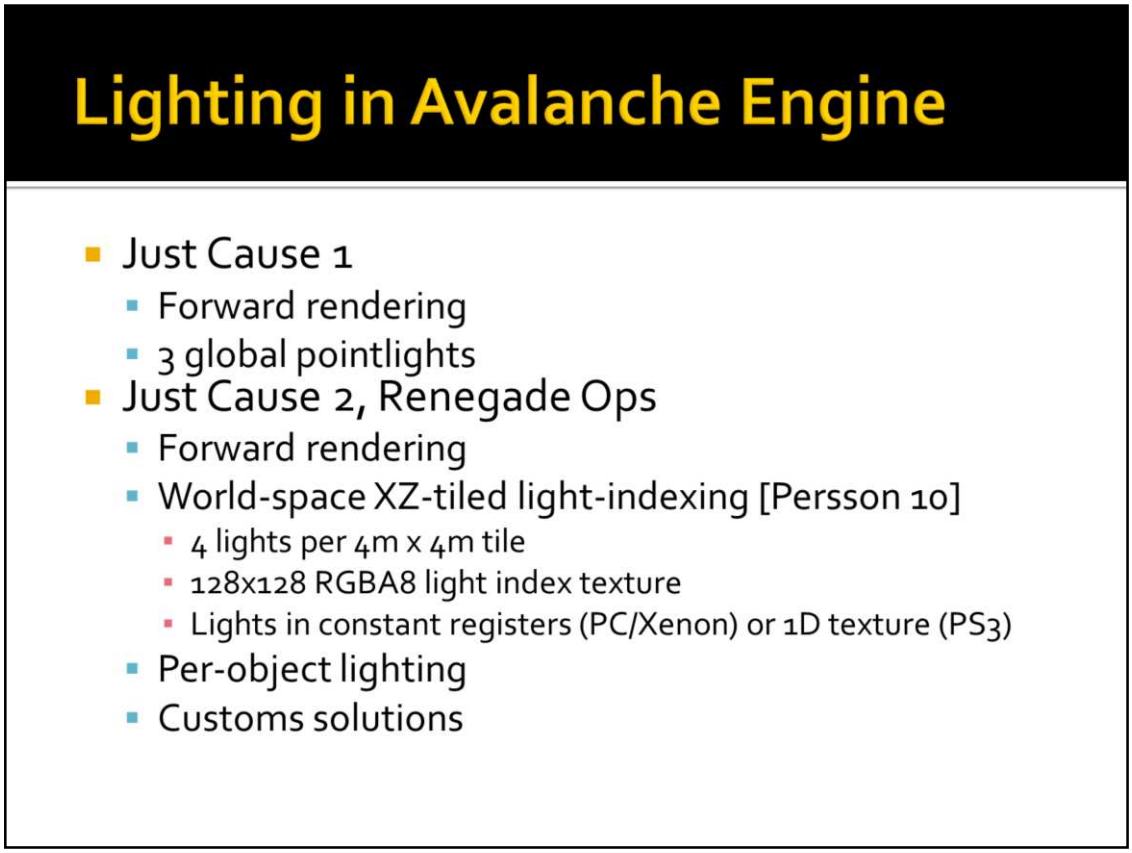
- The first part then is going to be presented by me.
- In this part we will provide an overview of techniques that have been, and still are, used in real-time shading.
- The main focus will be on clustered shading, as this is the most advanced and efficient technique today.

What's new?

- Preaching the Clustered gospel for two years
  - Nordic Game 2013
  - SIGGRAPH 2013
  - CEDEC 2013
  - SIGGRAPH Asia 2014
- Avalanche Studios still using it in production
  - Very happy with it
  - All old lighting paths removed
  - Game still unannounced …
- Interest in Clustered Shading increasing
  - Intel samples for PC and Android [Intel 14]
  - Gets name-dropped a lot by game developers
    - Although few have actually implemented it yet

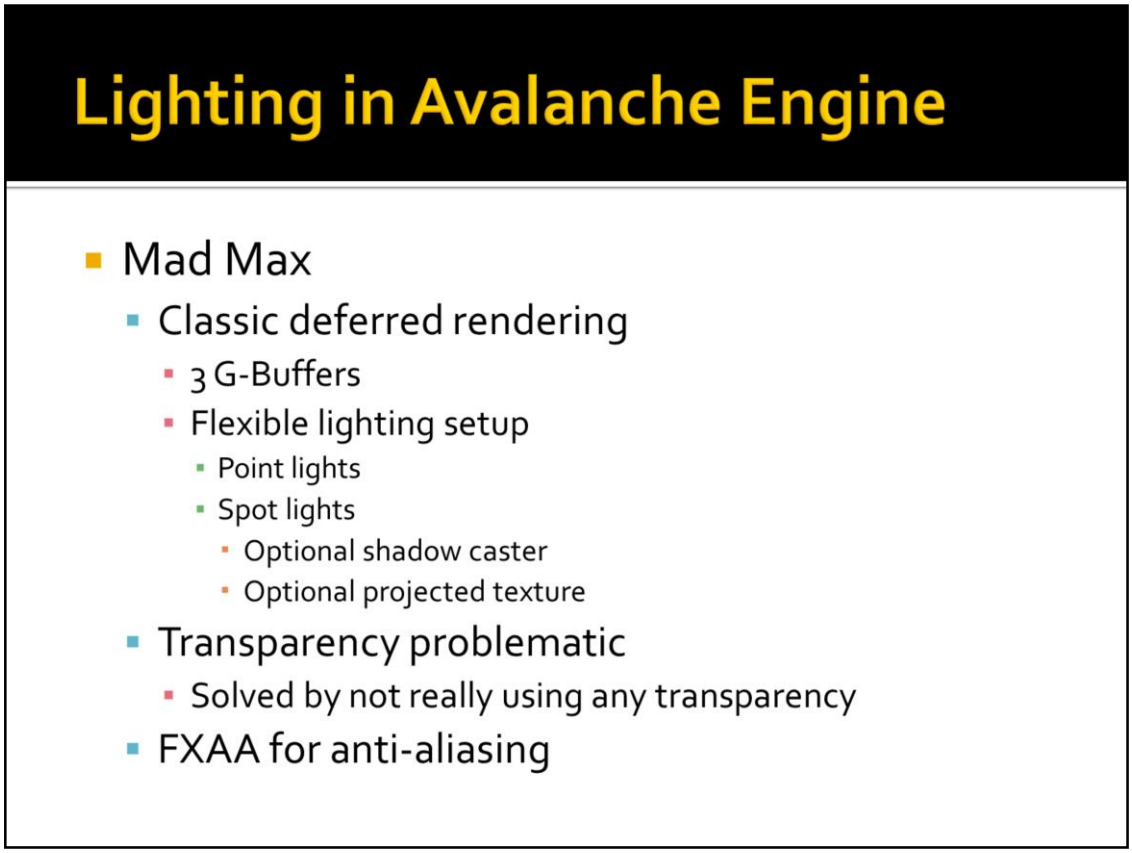
Agenda

- History of lighting in the Avalanche Engine
- Why Clustered Shading?
- Adaptations for the Avalanche Engine
- Performance
- Future work



Just Cause 1 had 3 global pointlights. This meant that if, for instance, three streetlights were enabled and you fired your gun, one of the lights would shut off for the duration of the gun flash. Clearly, this solution was hardly ideal.

For Just Cause 2 we switched to a world-space 2D tiled solution where light indexes were stored in texels. The technique has been described in detail in the article ”Making it Large, Beautiful, Fast, and Consistent: Lessons Learned Developing Just Cause 2” in GPU Pro. This technique was actually in some ways similar to clustered shading, although much more limited and designed around DX9 level hardware. It worked reasonably well on platforms with decent dynamic branching, such as PC and Xenon, whereas the PS3 struggled. Ultimately this caused us to implement numerous workarounds to get PS3 running well, so that in the end this technique mostly ended up being a fallback option if the light count was too high for a specialized shader to work. The amount of specialized shaders also became quite a bit of a maintenance problems, and figuring out the light count a performance issue on the CPU side.



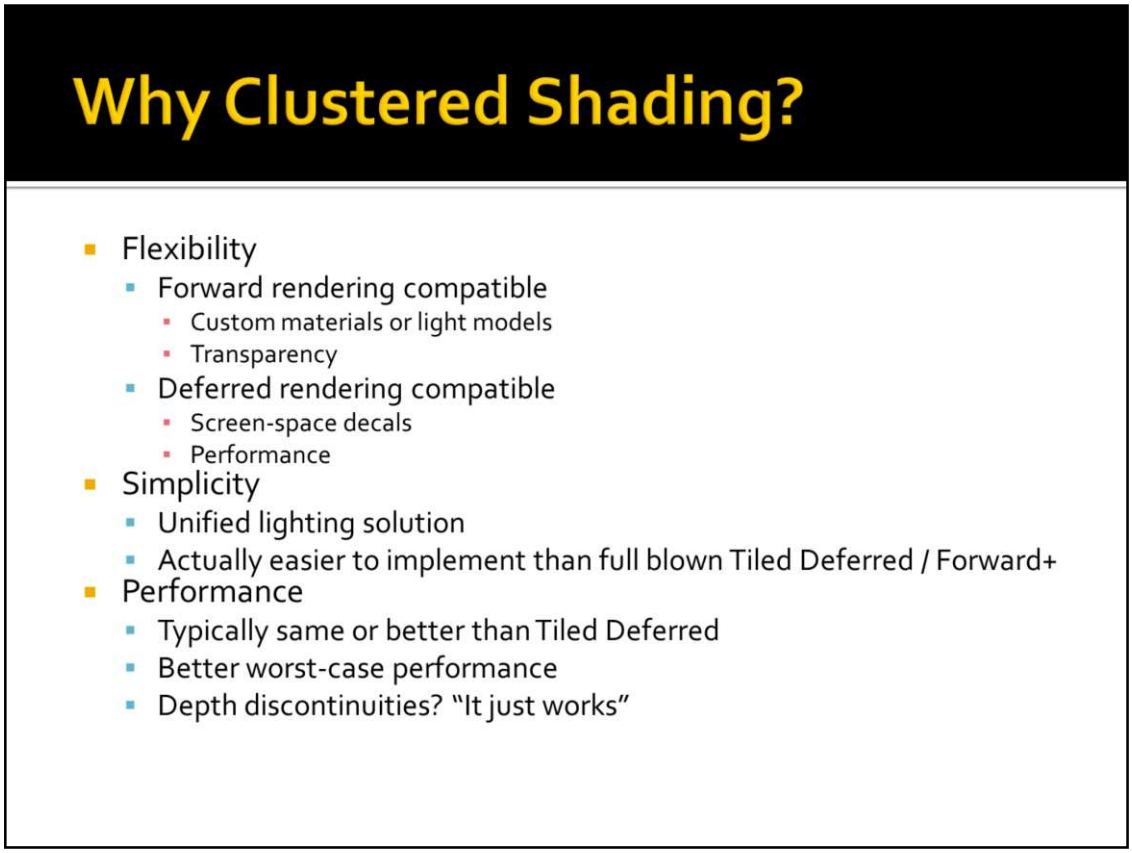
After Just Cause 2 we ended going the deferred shading route, initially using classic deferred. This worked relatively well for last generation console hardware and allowed us to support many more lights, different light types, shadow casting dynamic lights etc. This was great, but naturally we also got all the downsides of deferred shading, such as problems with transparency, problems with custom material or lighting models, as well as large increase in memory consumption. Initially we supported MSAA, but ultimately we dropped it in favor of FXAA for performance and memory reasons.

Unfortunately, the old forward pass also had to stick around for transparency to work to some extent, although it only ever supported pointlights and the lighting didn't quite match the much more sophisticated deferred pass. For Mad Max we ultimately moved away from supporting transparency with lighting because of its problems with deferred, plus that the game environment has very little need for transparency anyway beyond particle effects. But for other projects where transparency might be desirable we started looking into alternatives, especially with a new generation consoles on the horizon at the time.

Tiled Deferred Shading and Forward+ (Tiled Forward Shading) are production proven and has shipped in real games, but they come with a bunch of drawbacks. Tiled deferred offers better performance than classic deferred, but doesn't really solve any of our problems since all drawbacks of classic deferred stays around. In addition, it also imposes a new restriction in that all lights, and consequently shadow buffers, are now required up-front. However, this is a property it shares with all other techniques, including Forward+ and Clustered Shading, and even our old forward solution from JC2.

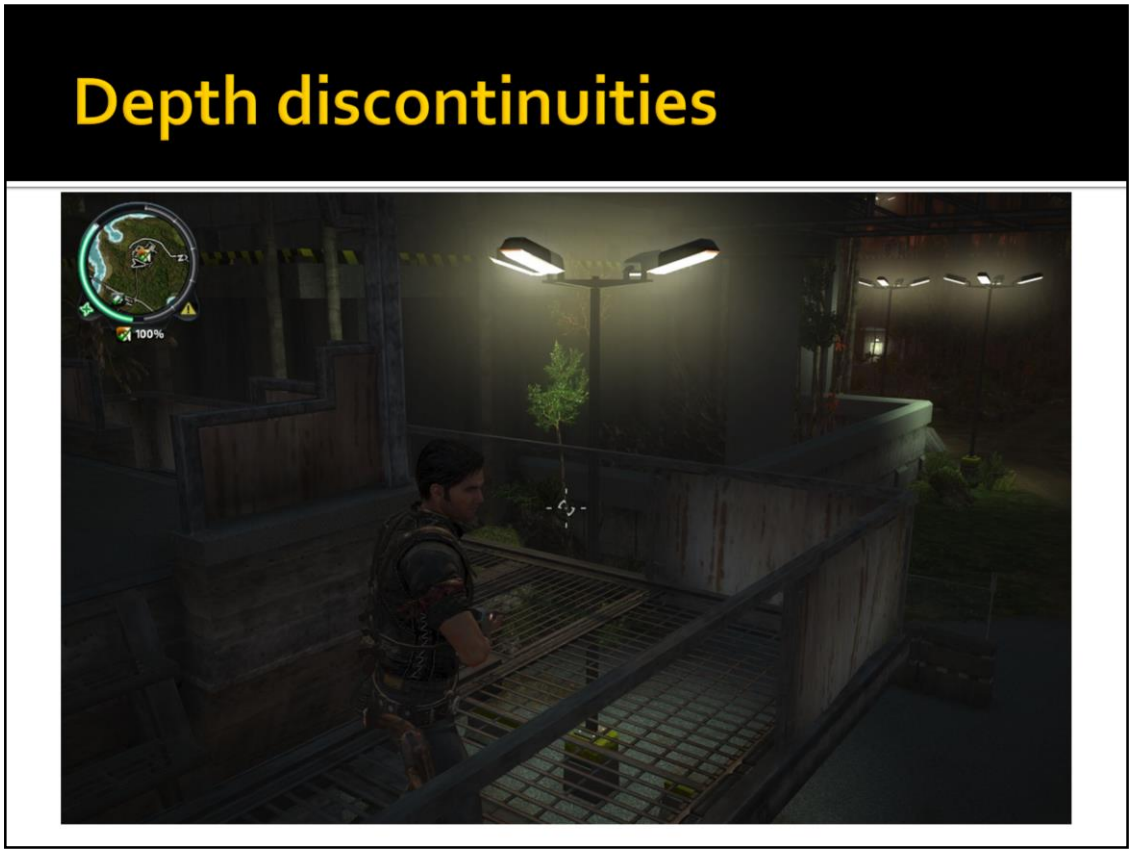
Forward+ has the advantage of working well with MSAA without hassles; however, while it can be made to work with transparency, it requires an extra pass, including another round of pre-z. The requirement of a full pre-z pass for this technique to work made this a non-starter for us. We didn't bother implementing it for evaluation purposes as a full pre-z pass is not an option for us. We did at one point have a fairly complete pre-z pass in Just Cause 2, but over the development the pre-z pass was continuously trimmed until very little remained. The additional overhead just didn't pay off, and the large increase in draw-call count was problematic. After we got a decent occlusion culling system in place there were very few cases pre-z did not, in fact, result in a performance drop. Pre-z is now only enabled on a handful of things specifically marked for pre-z by content creators, and a few code-driven systems that need it for other reasons.

Clustered Shading has the advantage of not requiring a pre-z pass, even in its forward incarnation, while working well with MSAA and transparency out of the box with no particular tricks or hacks. It has at the point of this writing to our knowledge not shipped in any real games so far, but it has been in production at Avalanche Studios since January 2013 and has so far worked really well for us and we expect it to make it all the way to shipping.



Clustered Shading is really decoupled from the choice between deferred or forward rendering. It works with both, so you're not locked into one or the other. This way you can make an informed choice between the two approaches based on other factors, such as whether you need custom materials and lighting models, or need deferred effects such as screen-space decals, or simply based on performance.

The two tiled solutions need quite a bit of massaging to work reasonable well in all situations, especially with large amounts of depth discontinuities. There are proposed solutions that mitigate the problem, such as 2.5D culling, but they further complicate the code. For Clustered Shading it just falls out automatically and depth discontinuities do not cause performance problems. This allows Clustered Shading to maintain a more stable frame-rate regardless of scene depth complexity.



Since we can't show any screenshots of what we're working on at this point, the problem will be illustrated with Just Cause 2. This is what I got when I just launched my last save-game from the retail game. I didn't have to go look for a problematic area, in fact, it was right there in front of my face. This shows how common these scenes actually are in real games, and certainly so in the games that we make.

Depth discontinuities

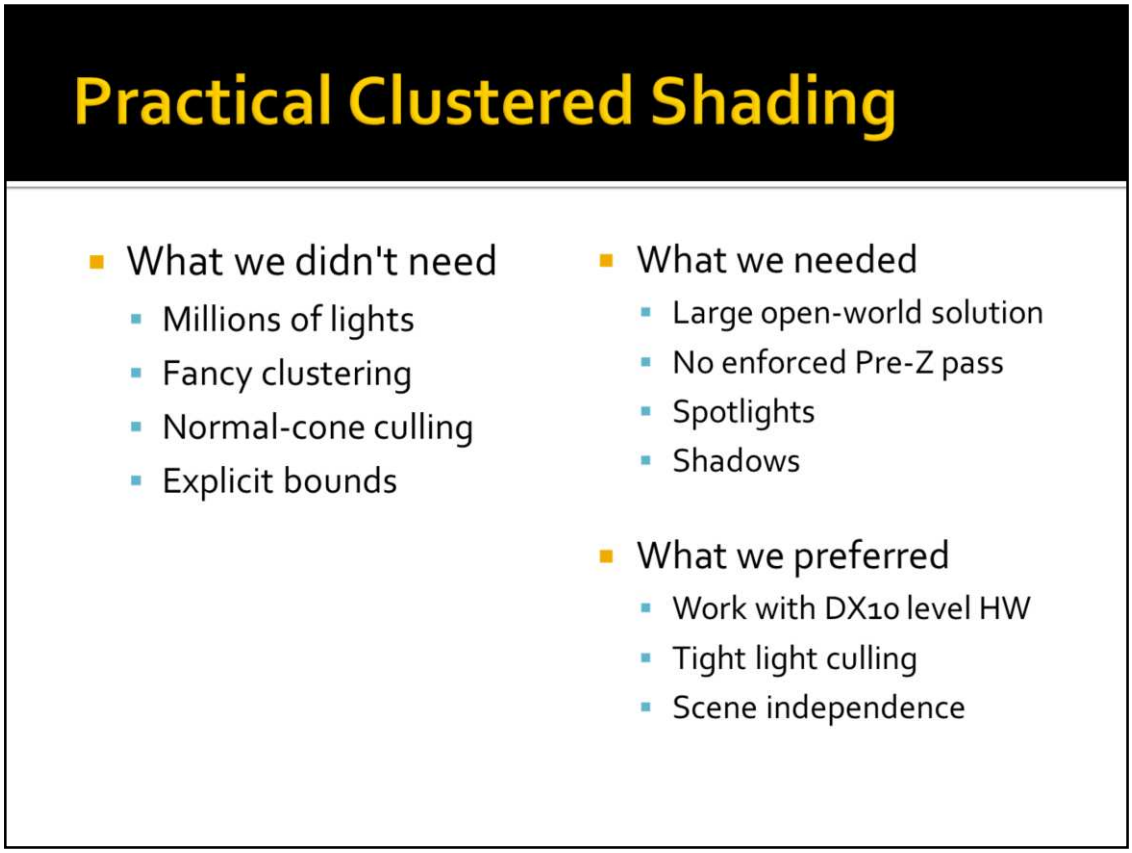
Here a number of large depth differences have been manually painted over the image to illustrate where you might expect a problem for tiled shading techniques. As you can see, they are fairly common and affect a fairly large part of the screen. The lattice the player is standing on is a typical example of a common piece of game art that would cause problems for tiled techniques. Foliage is another typical source of pain.

Depth discontinuities

If you've played Just Cause 2, you know that this sort of scenario is actually quite common, and obviously problematic from a depth discontinuity point of view.

Depth discontinuities

And here we can see that in this scene, pretty much the whole screen is filled of areas that would be much more expensive to shade with the tiled techniques than with clustered.

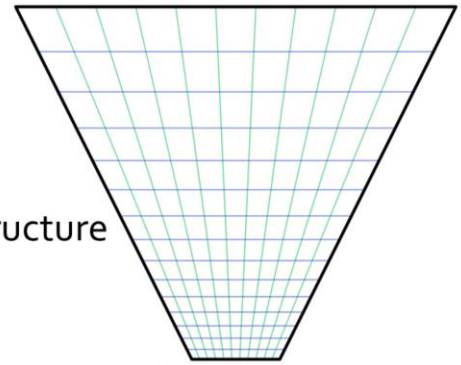


The original paper [Olsson et. al 12] was written by academics, and naturally the direction of their research doesn't match 100% with the requirements of the a game engine. We don't have millions of tiny lights, but between hundreds and thousands of mostly artist placed lights, that are on a human scale. This meant that tight culling, so as to not add lights to more clusters than necessary, became more important to us. The higher-order clustering options the paper explored (and also largely rejected) were also something that we didn't expect to work for us. Deriving the explicit cluster bounds was something that could be interesting, but we found that sticking to implicit bounds simplified the technique, while also allowing the light assignment to run on the CPU. This enables DX10 level GPU compatibility, which may be important given that (as of this writing) 24% on Steam are still on a DX10 GPU. In addition, this gives us scene independence. This means that we don't need to know what the scene looks like to fill in the clusters, and this also allows us to evaluate light at any given point in space, even if it's floating in thin air. This could be relevant for instance for ray-marching effects.

The paper only explored pointlights, whereas we need spotlights as well. We also needed a shadow solution, which the original paper also did not explore. However, Olsson et. al. has since continued their research and have now an interesting shadow approach made for clustered shading. We have however stuck with our own simpler approach. Finally, our games are massively large while still being played on human scale, resulting in a depth span from very near to very far, which required some extra fiddling to get rolling with clustered shading.

The Avalanche solution

- Still a deferred shading engine
  - But unified lighting solution with forward passes
- Only spatial clustering
  - 64x64 pixels, 16 depth slices
- CPU light assignment
  - Works on DX10 HW
  - Allows compacter memory structure
- Implicit cluster bounds only
  - Scene-independent
  - Deferred pass could potentially use explicit



We are still using a deferred engine, but we could change to forward at any time should we decide that to be better. The important part is, however, that the transparency passes can now use the same lighting structure as the deferred passes, making it a unified lighting solution. Since we are still using deferred, and thus obviously have a complete depth buffer once we get to the deferred lighting pass, we could potentially use explicit bounds there. We still haven't explored that opportunity, but it's an option. It's unclear if computing the explicit bounds, plus an extra round of culling, is going to be outweighed by potentially faster light evaluation.

Currently we are using 64x64 screen-space tiles, and 16 depth slices. This is most likely going to change, primarily because currently the tiles are currently fairly long and thin, and this is not optimal for a culling, in particular for spotlights. We have been experimenting with other setups, such as 128x128 and 32 depth slices. This created more cubical shaped clusters and helped with culling, which helped with culling, especially for spotlights. Another option we have considered, but not yet explored, is to not base it on pixel count, but simply divide the screen into a specific number of tiles regardless of resolution. This may reduce coherency on the GPU side somewhat in some cases, but would also decouple the CPU workload from the GPU workload and allow for some useful CPU side optimizations if the tile counts are known at compile time.

The Avalanche solution

- Exponential depth slicing
  - Huge depth range! [0.1m – 50,000m]
    - Default list
      - [0.1, 0.23, 0.52, 1.2, 2.7, 6.0, 14, 31, 71, 161, 365, 828, 1880, 4270, 9696, 22018, 50000]
      - Poor utilization
    - Limit far to 500
      - We have a "distant lights" systems for light visualization beyond that
      - [0.1, 0.17, 0.29, 0.49, 0.84, 1.43, 2.44, 4.15, 7.07, 12.0, 20.5, 35, 59, 101, 172, 293, 500]
    - Special near 0.1 – 5.0 cluster
      - Tweaked visually from player standing on flat ground
      - [0.1, 5.0, 6.8, 9.2, 12.6, 17.1, 23.2, 31.5, 42.9, 58.3, 79.2, 108, 146, 199, 271, 368, 500]

We are using exponential depth slicing, much like in the paper. There is nothing dictating that this is what we have to use, or for that matter that it is the best or most optimal depth slicing strategy; however, the advantage is that the shape of the clusters remain the same as we go deeper into the depth. On the other hand, clusters get larger in world space, which could potentially result in some distant clusters containing a much larger amount of lights. Depending on the game, it may be worth exploring other options.

Our biggest problem was that our depth ratio is massive, with near plane as close as 0.1m and far plane way out on the other side of the map, at 50,000m. This resulted in poor utilization of our limited depth slices, currently 16 of them. The step from one slice to the next is very large. Fortunately, in our game we don't have any actual light sources beyond a distance of 500m. So we simply decided to keep our current distant light system for distances beyond 500m and limit the far range for clustering to that.

This improved the situation notably, but was still not ideal. We still burnt half of our slices on the first 7 meters from the camera. Given how our typical scenes look like, that's likely going to be mostly empty space in most situations. So to improve the situation, we made the first slice special and made that go from near plane to an arbitrary visually tweaked distance, currently 5m. This gave us much better utilization.

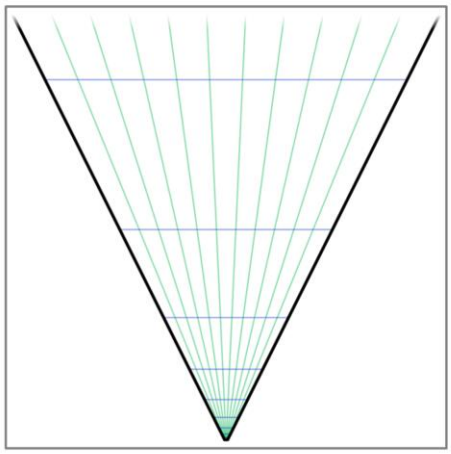
The Avalanche solution

- Separate distant lights system

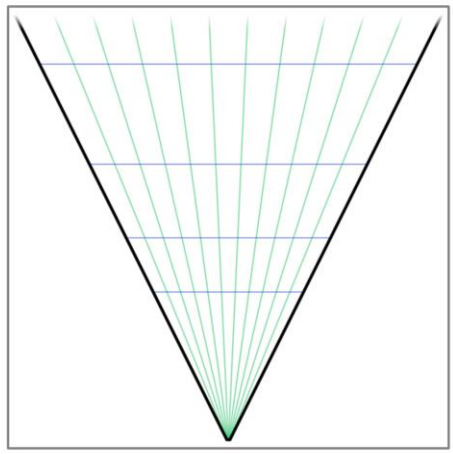
This illustrates our distant light system, which has been around since Just Cause 2. In this screenshot there are likely no actual lights enabled since we're far from civilization on top of a mountain, except perhaps our fake "night light" that slightly illuminates the area around the player at night to help game-play a bit in the darkness. Everything in the distance though, while representing actual artist placed lights, the actual light sources aren't loaded at this distance. They are simply stored as a very compact list of point sprites, resident in memory at all time, and which is very cheap to render. We are at this point still using the same forward rendering solution here as in Just Cause 2, but one option now that we are using deferred is to actually compute real lighting under those sprites instead of just a putting a blob from a texture there.

The Avalanche solution

- Default exponential spacing



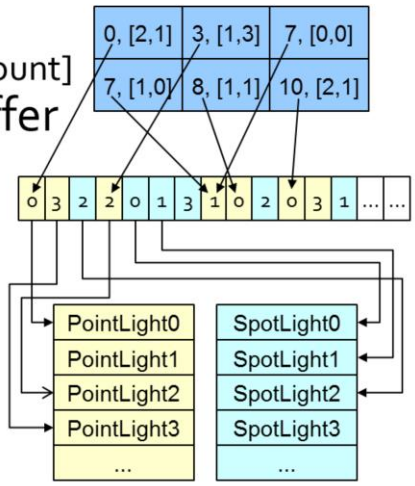
- Special near cluster



This illustrates the benefit of the special near cluster. Less slices are wasted, and the cluster shapes aren't quite as long and thin.

Data structure

- Cluster "pointers" in 3D texture
  - R32G32_UINT
    - R=Offset
    - G=[PointLightCount, SpotLightCount]
- Light index list in texture buffer
  - R16_UINT
  - Tightly packed
- Light & shadow data in CB
  - PointLight: 2 x float4
  - SpotLight: 3 x float4



Given a screen position and a depth value (whether from a depth buffer or the rasterized depth in a forward pass) we start by looking up the cluster from a 3D texture. Each texel represents a cluster and its light list. The red channel gives us an offset to where the light list starts, whereas the green channel contains the light counts. The light lists are then stored in a tightly packed lists of indexes to the lights. The actual light source data is stored as arrays in a constant buffer.

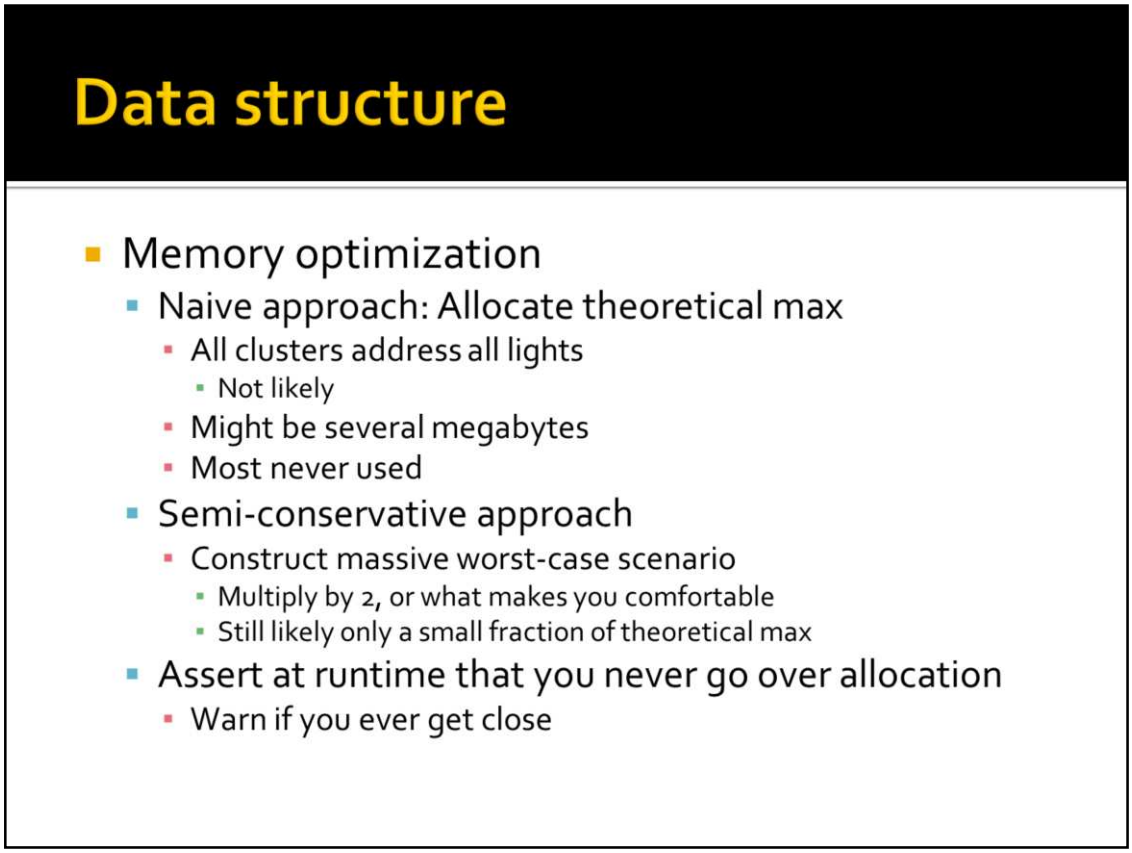
All in all the data structure is very compact. In a typical artists lit scene it may be around 50-100kb of data to upload to the GPU every frame.

Shader

```
int3 tex_coord = int3(In.Position.xy, 0);                  // Screen-space position ...
float depth = Depth.Load(tex_coord);                       // ... and depth

int slice = int(max(log2(depth * ZParam.x + ZParam.y) * scale + bias, 0)); // Look up cluster
int4 cluster_coord = int4(tex_coord >> 6, slice, 0);       // TILE_SIZE = 64

uint2 light_data = LightLookup.Load(cluster_coord);        // Fetch light list
uint light_index = light_data.x;                           // Extract parameters
const uint point_light_count = light_data.y & 0xFFFF;
const uint spot_light_count  = light_data.y >> 16;

for (uint pl = 0; pl < point_light_count; pl++) {          // Point lights
    uint index = LightIndices[light_index++].x;

    float3 LightPos = PointLights[index].xyz;
    float3 Color    = PointLights[index + 1].rgb;
    // Compute pointlight here ...
}

for (uint sl = 0; sl < spot_light_count; sl++) {           // Spot lights
    uint index = LightIndices[light_index++].x;

    float3 LightPos = SpotLights[index].xyz;
    float3 Color    = SpotLights[index + 1].rgb;
    // Compute spotlight here ...
}
```

This shows the shader code for rendering with this data structure. The input is just the screen-space position and depth. This shows a deferred pass where depth comes from a texture, but in a forward pass the second line of code would simply use In.Position.z instead. Everything else would be identical, which shows how easily this technique adapts to either deferred or forward.

The ZParam.xy here contains the same parameters that you would use to compute a linear depth from a Z-buffer value, except I eliminated the division since that just becomes a negative under the logarithm, i.e. log2(1/(z*a+b)) = log2(z*(-a)+(-b)).

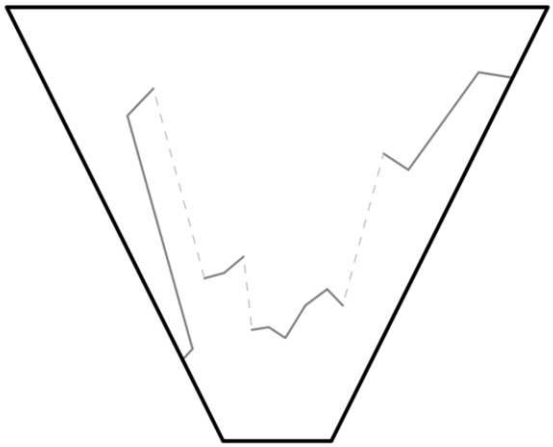


The light list could theoretically become huge. Say you have a total of 30*17*16 clusters at 1080p, and allow up to 256 lights per cluster, that would need 4MB, which with double-buffering (because it's updated from the CPU) means you'll need 8MB. Perhaps not a problem on next-gen, but hardly ideal, and who knows how many times these numbers will be bumped before you ship.

Normally, not every light affects every cluster in a scene. In fact, it's extremely rare that you get even remotely close to that. So we constructed a somewhat plausible worst-case scenario with loads of large lights jammed in front of the player and recorded the max utilization ever encountered. Then multiplied up that for some extra margin. Even after that, the resulting buffer size we needed to allocate was far smaller. Naturally though, if you go down this path, it's clearly important to add runtime assertions and warnings to make sure you don't ever go above what you actually have allocated. Done correctly, at worst you would have artifacts for that extreme frame where a thousand nukes blew up in the player's face.

Clustering and depth

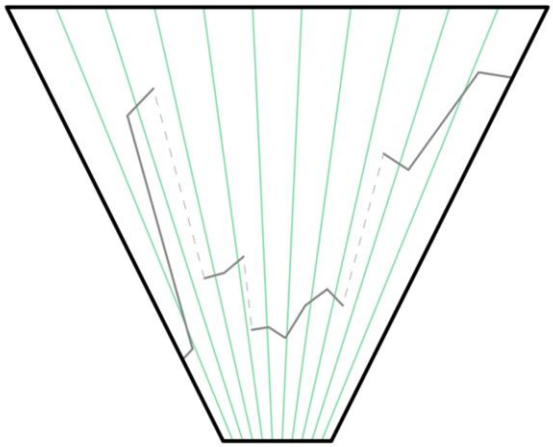
- Sample frustum with depths



Let's discuss the problem of depth discontinuities and illustrate how clustered shading solves it. Here's a sample frustum with some depth values, including a few discontinuities.

Clustering and depth

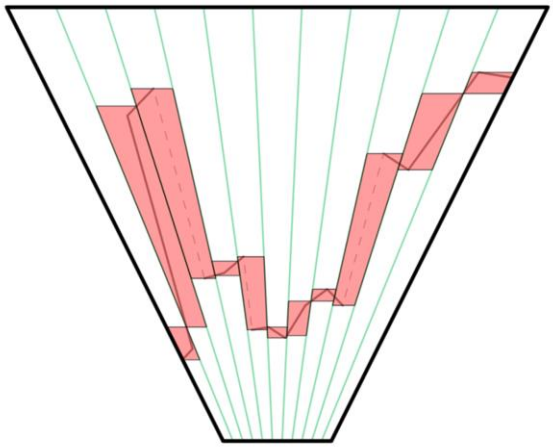
- Tiled frustum



Here we added the tiles.

Clustering and depth

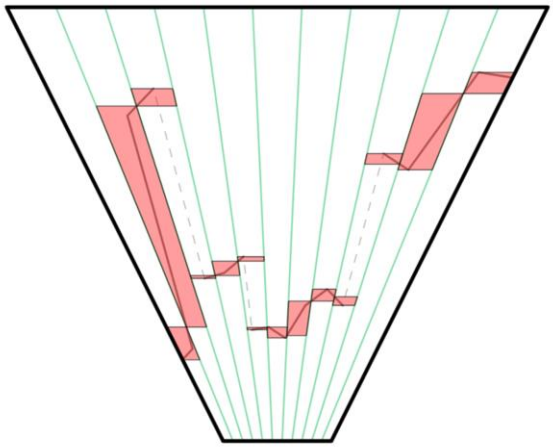
- Depth ranges for Tiled Deferred / Forward+



And this is the depth ranges you would get for a plain tiled shading algorithm. Clearly some ranges are fairly large.

Clustering and depth

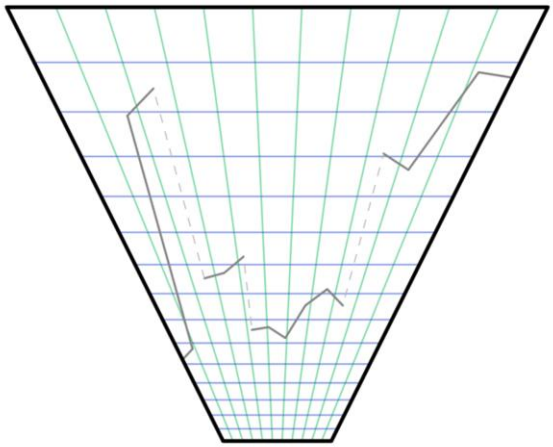
- Depth ranges for Tiled Deferred / Forward+ with 2.5D culling [Harada 12]



With 2.5D culling the situation is notably improved. Now lights in the discontinuity area is not included. However, we do pay the full cost lights at both ends for both sides of the discontinuity. Also note that one very long depth range remains. This is because it's not discontinuous, it's a continuous slope. This situation would happen if you look down a hallway, or the ground plane, or moderate large surface at a grazing angle.

Clustering and depth

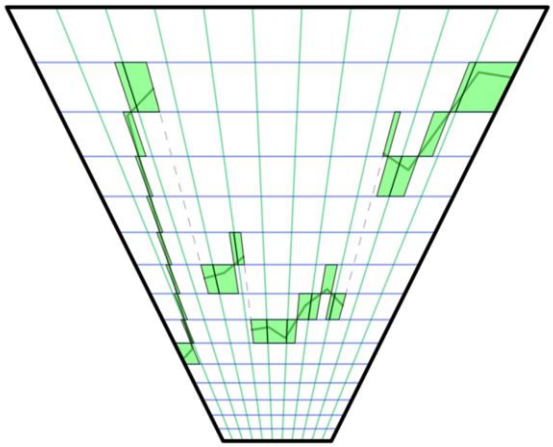
- Clustered frustum



Now let's look at a clustered frustum.

Clustering and depth

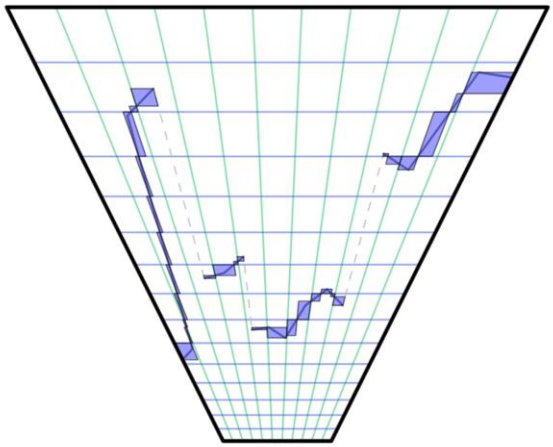
- Implicit depth ranges for clustered shading



These are the depth ranges that we will need to consider. Note that we are paying for exactly one cluster's depth at any given point.

Clustering and depth

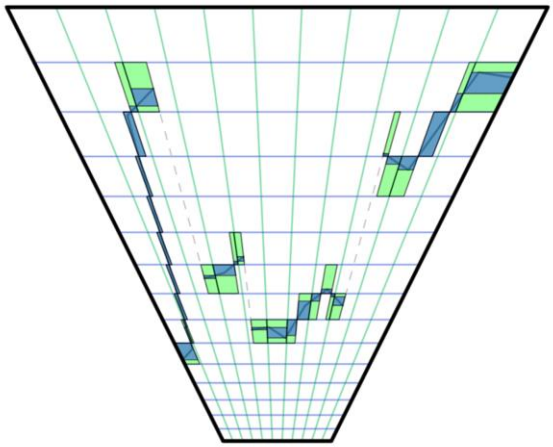
- Explicit depth ranges for clustered shading



If we go to explicit cluster bounds, the situation is even further improved, although in practice there may not be a huge difference between a fairly small range and an even smaller range, depending on the typical size of light sources.

Clustering and depth

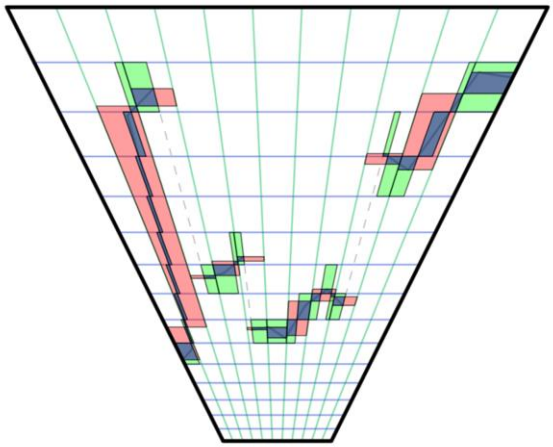
- Explicit versus implicit depth ranges



Here we see the improvement from implicit bounds to explicit.

Clustering and depth

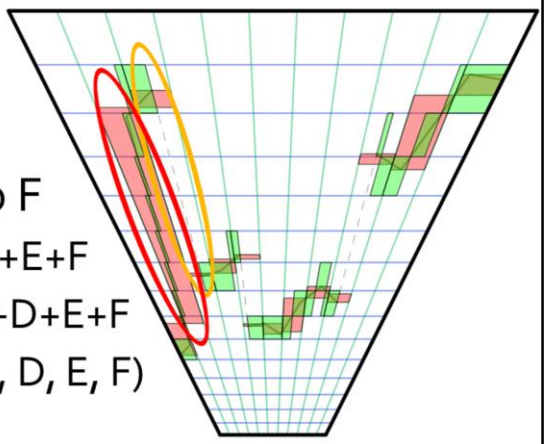
- Tiled vs. implicit vs. explicit depth ranges



And here all techniques are compared. As you can see, explicit clustered is always the tightest. However, there are definitively areas where tiled with 2.5D culling is tighter than implicit cluster bounds. So in scenes with little depth complexity tiled could very well be faster. However, implicit clustered bounds does not have any areas that are extremely bad, regardless of depth complexity, and would thus perform more consistently. Most importantly, it's worst case performance is much better than tiled.

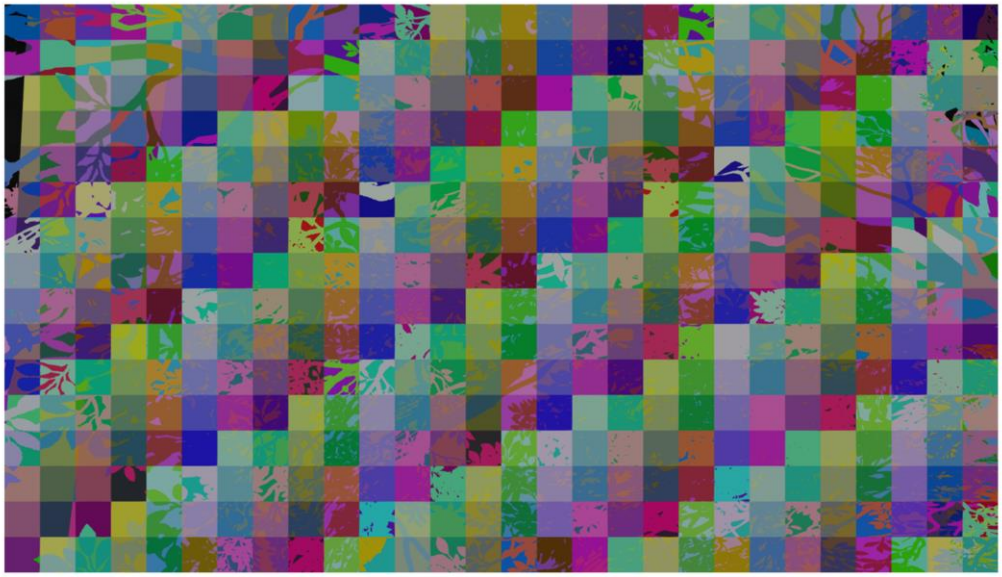
Wide depths

- Depth discontinuity range A to F
  - Default Tiled: A+B+C+D+E+F
  - Tiled with 2.5D: A + F
  - Clustered: ~max(A, F)
- Depth slope range A to F
  - Default Tiled: A+B+C+D+E+F
  - Tiled with 2.5D: A+B+C+D+E+F
  - Clustered: ~max(A, B, C, D, E, F)



Here we can see the impact of adding 2.5D culling to a tiled technique. While it helps in the discontinuity case (although does not reach clustered's performance), it doesn't help much or at all in a depth slope situation.

Data coherency



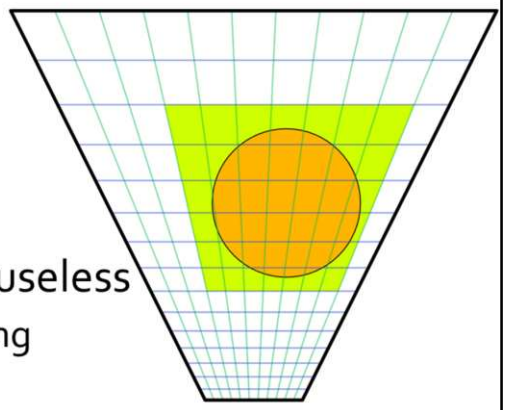
So the difference between tiled and clustered is that we pick a light list on a per-pixel basis instead of per-tile, depending on which cluster we fall within. Obviously though, in a lot of cases nearby pixels will choose the same light list, in particular neighbors within the same tile on a similar depth. If we visualize what light lists were chosen, we can see that there are a bunch of different paths taken beyond just the tile boundaries. A number of depth discontinuities from the foliage in front of the player gets clearly visible. This may seem like a big problem, but here we are only talking about fetching different data. This is not a problem for a GPU, it's something they do all the time for regular texture fetches, and this is even much lower frequency than that.

Branch coherency

The thing you might worry about though is divergent branches. However, despite fetching different light lists from pixel to pixel, the situation is not nearly as bad as you might expect from the previous picture. Chances are that the light lists look fairly similar. If you have one light lists with 5 lights and another with 5 lights (that are not necessarily the same as the other ones), branching will still be 100% coherent. You may pay a small overhead from the ideal when the lists have different light count, but that is typically going to be a relatively small overhead. In the worst-case scenario (no coherency at all), the amount of shading essentially boils down to what tiled shading has to shade.

Culling

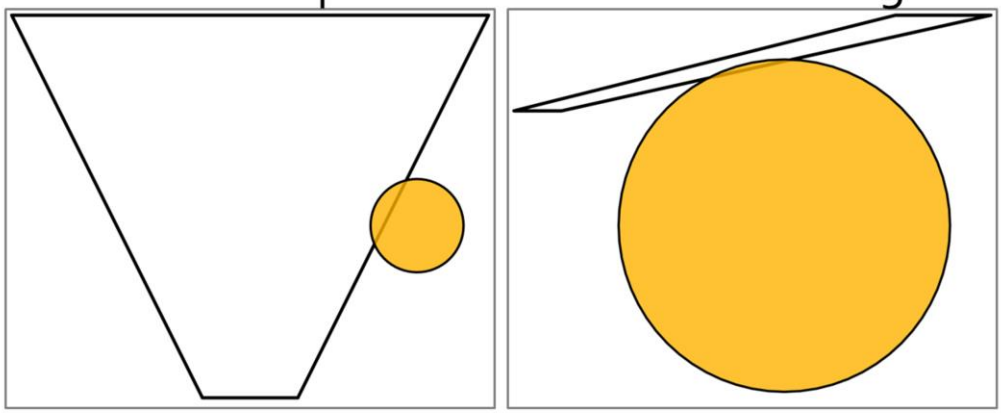
- Want to minimize false positives
- Must be conservative
  - But still tight
  - Preferably exact
    - But not too expensive
    - Surprisingly hard!
- 99% frustum culling code useless
  - Made for view-frustum culling
    - Large frustum vs. small sphere
    - We need small frustum vs. large sphere
  - Sphere vs. six planes won't do



Our light sources are typically artist placed, scaled for human environments in an outdoor world, so generally speaking from meters to tens of meters. So a light source generally intersects many clusters. The typical sphere-frustum tests that you can find online are not suitable for this sort of culling. They are made for view-frustum culling and based on the assumption that the frustum typically is much larger than the sphere, which is the opposite of what we have here. Typically they simply test sphere vs plane for each six planes of the frustum. This is conservative, but lets through spheres that aren't completely behind any of the planes, such as in the frustum corners. The result you get is that green rectangle, or essentially a "cube" of clusters around the light. But that's also the first thing we compute. We simply compute the screen-space and depth extents of the light analytically first, so this test doesn't actually help anything at all after that.

Culling

- Your mental picture of a frustum is wrong!



Most frustum culling code is written with the scenario on the left in mind. We need to handle the scenario on the right.

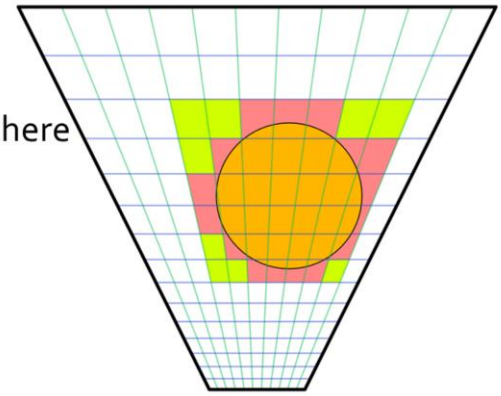
Culling

- "Fun" facts:
  - A sphere projected to screen is not a circle
  - A sphere under projection is not a sphere
  - The widest part of a sphere on screen is not aligned with its center
  - Cones (spotlights) are even harder
- Frustums are frustrating (pun intended)
- Workable solution:
  - Cull against each cluster's AABB

One way to go about frustum culling is testing all planes, all edges and all vertices. This would work, but be too costly to outweigh the gains from fewer false positives. A fast, conservative but relatively tight solution is what we are looking for. There are many approaches that seem fitting, but there are also many complications, which has ultimately thrown many of our attempts into the garbage bin. One relatively straightforward approach is to cull against the cluster's AABB. This is fast and gives fairly decent results, but it's possible to do better.

Pointlight Culling

- Our approach
  - Iterative sphere refinement
    - Loop over z, reduce sphere
    - Loop over y, reduce sphere
    - Loop over x, test against sphere
  - Culls better than AABB
    - Similar cost
    - Typically culling 20-30%

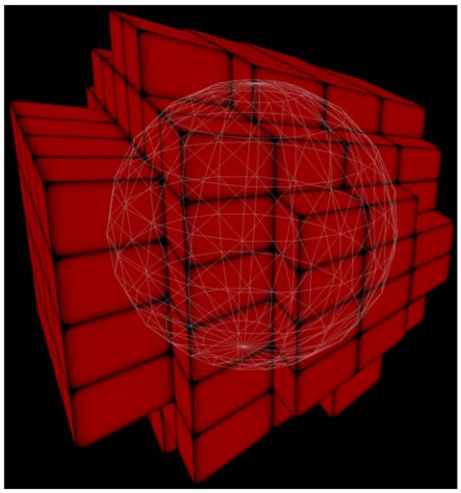


Starting with the ”cube” of clusters around the light, in our outer loop we iterate over the slices in z direction. We intersect the sphere with the slice where it is the widest. This results in a circle of a smaller radius than the original sphere, we thus continue in the y direction using a sphere of this smaller radius and the circle’s midpoint. In the center slice we simply proceed with the original sphere. We repeat this procedure in y and have an even smaller sphere. Then in the inner loop we do plane vs. sphere tests in x direction to get a strip of clusters to add the light to.

To optimize all the math we take advantage of the fact that in view-space, all planes will have components that are zero. A plane in the x direction will have zero y and offset, y direction has zero x and offset, and z-direction is basically only a z offset.

The resulting culling is somewhat tighter than a plain AABB test, and costs about the same. Where AABB culls around 15-25%, this technique culls around 20-30% from the “cube” of clusters.

Pointlight Culling



Here's the result visualized in 3D.

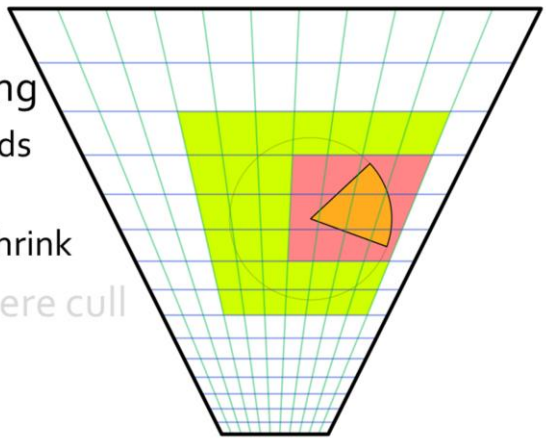
Culling pseudo-code

```
for (int z = z0; z <= z1; z++) {
    float4 z_light = light;
    if (z != center_z) {          // Use original in the middle, shrunken sphere otherwise
        const ZPlane &plane = (z < center_z)? z_planes[z + 1] : -z_planes[z];
        z_light = project_to_plane(z_light, plane);
    }
    for (int y = y0; y < y1; y++) {
        float3 y_light = z_light;
        if (y != center_y) {      // Use original in the middle, shrunken sphere otherwise
            const YPlane &plane = (y < center_y)? y_planes[y + 1] : -y_planes[y];
            y_light = project_to_plane(y_light, plane);
        }
        int x = x0;               // Scan from left until with hit the sphere
        do { ++x; } while (x < x1 && GetDistance(x_planes[x], y_light_pos) >= y_light_radius);

        int xs = x1;              // Scan from right until with hit the sphere
        do { --xs; } while (xs >= x && -GetDistance(x_planes[xs], y_light_pos) >= y_light_radius);

        for (--x; x <= xs; x++)   // Fill in the clusters in the range
            light_lists.AddPointLight(base_cluster + x, light_index);
    }
}
```

This shows the gist of the culling code.

Spotlight Culling

- Our approach
  - Iterative plane narrowing
    - Find sphere cluster bounds
    - In each six directions, do plane-cone test and shrink
  - Cone vs. bounding-sphere cull remaining "cube"

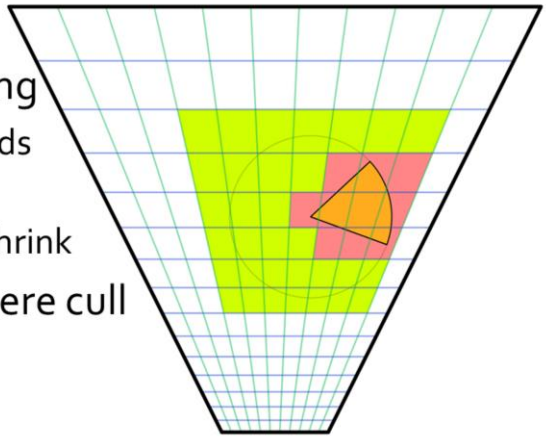


For spotlights we begin by finding the ”cube” of clusters around the light’s sphere, just like for pointlights, except this cube typically is much larger than necessary for a spotlight. However, this analytical test is cheap and goes a long way to limit the search space for following passes. Next we find a tighter ”cube” simply by scanning in all six directions, narrowing it down by doing plane-cone tests. There is likely a neat analytical solution here, but this seemed non-trivial. Given that the plane scanning works fine and is cheap we haven’t really explored that path.

Note that our cones are sphere-capped rather than flat-capped. That’s because the light attenuation is based on distance (as it should), rather than depth. Sphere-capped cones also generally behave much better for wide angles and doesn’t become extremely large as flat-capped cones can get.

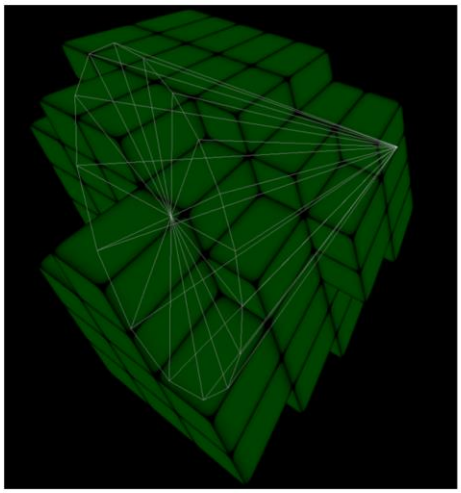
Spotlight Culling

- Our approach
  - Iterative plane narrowing
    - Find sphere cluster bounds
    - In each six directions, do plane-cone test and shrink
  - Cone vs. bounding-sphere cull remaining "cube"



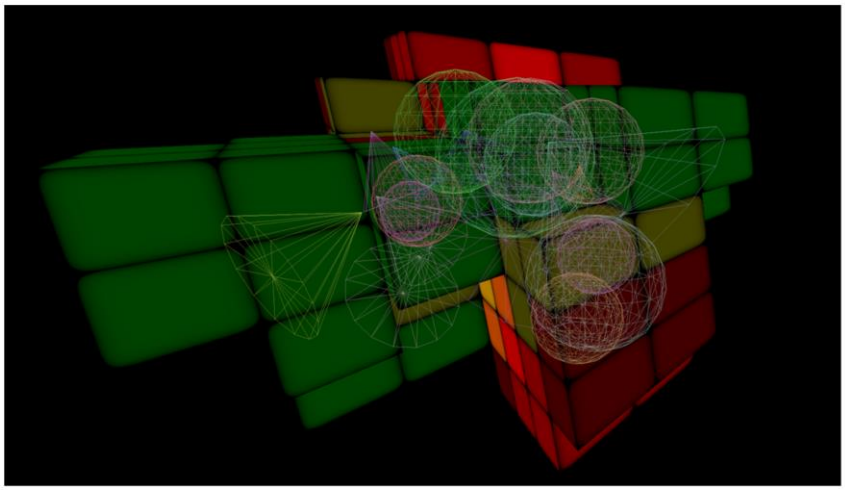
Finally, for the remaining ”cube” of clusters we cull each cluster with a sphere-capped cone vs. bounding sphere test. For this to work well we have to have relatively cubical shaped clusters, otherwise the bounding sphere becomes way oversized. Overall this technique results in a moderately tight culling that is good enough for us so far, although there is room for some minor improvement.

Spotlight Culling

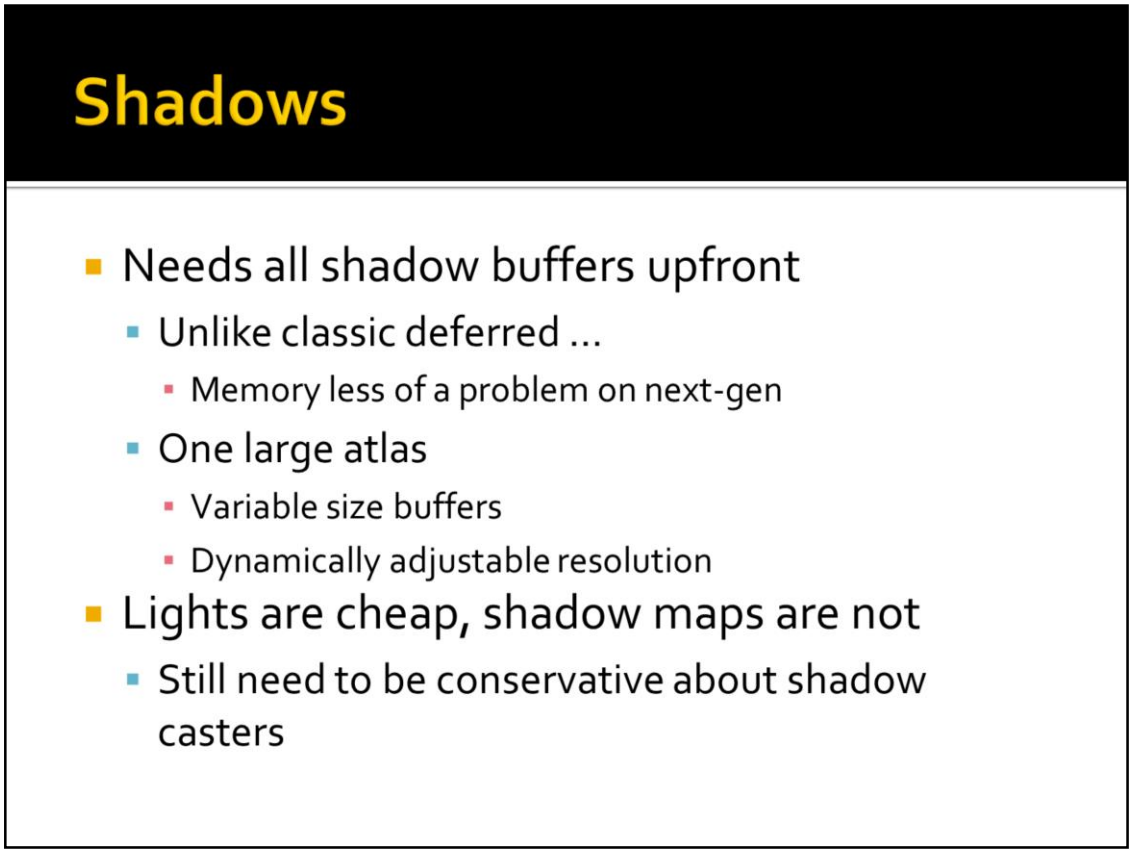


Here's the result visualized in 3D. Although our spotlights are sphere-capped, our debug visualization still draws them as flat-capped. That's why it might look like it's extending a bit outside the clusters.

Pointlights and spotlights



Here's the result with a handful of pointlights and spotlights enabled in a scene. The number of pointlights goes into red, and number of spotlights into green.



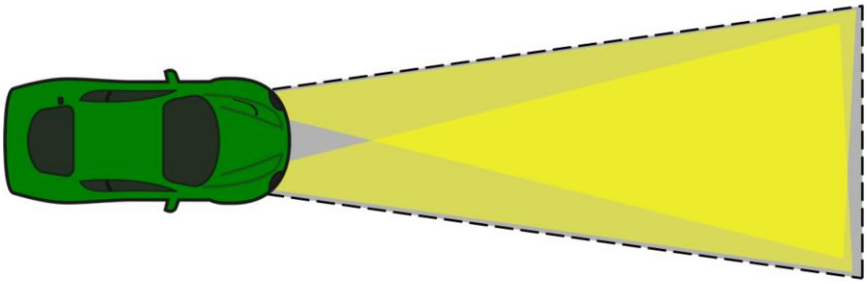
Classic deferred has the advantage that you can iterate light by light, and thus reuse resources such as shadow buffers in between. This saves some memory, which may be needed on current generation consoles. On PC and next-generation consoles this is not nearly as big a problem.

With the switch to clustered shading the cost of adding a new light to the scene is small. Artists can now be moderate ”wasteful” without causing much problems performance-wise. This is not true for rasterizing shadow buffers. They remain expensive, and relatively speaking going to be more expensive going forward since it’s often a ROP-bound process, and ROPs aren’t getting scaled up nearly as much as ALU. So we still need to be a bit conservative about how many shadow casting lights we add to the scene.

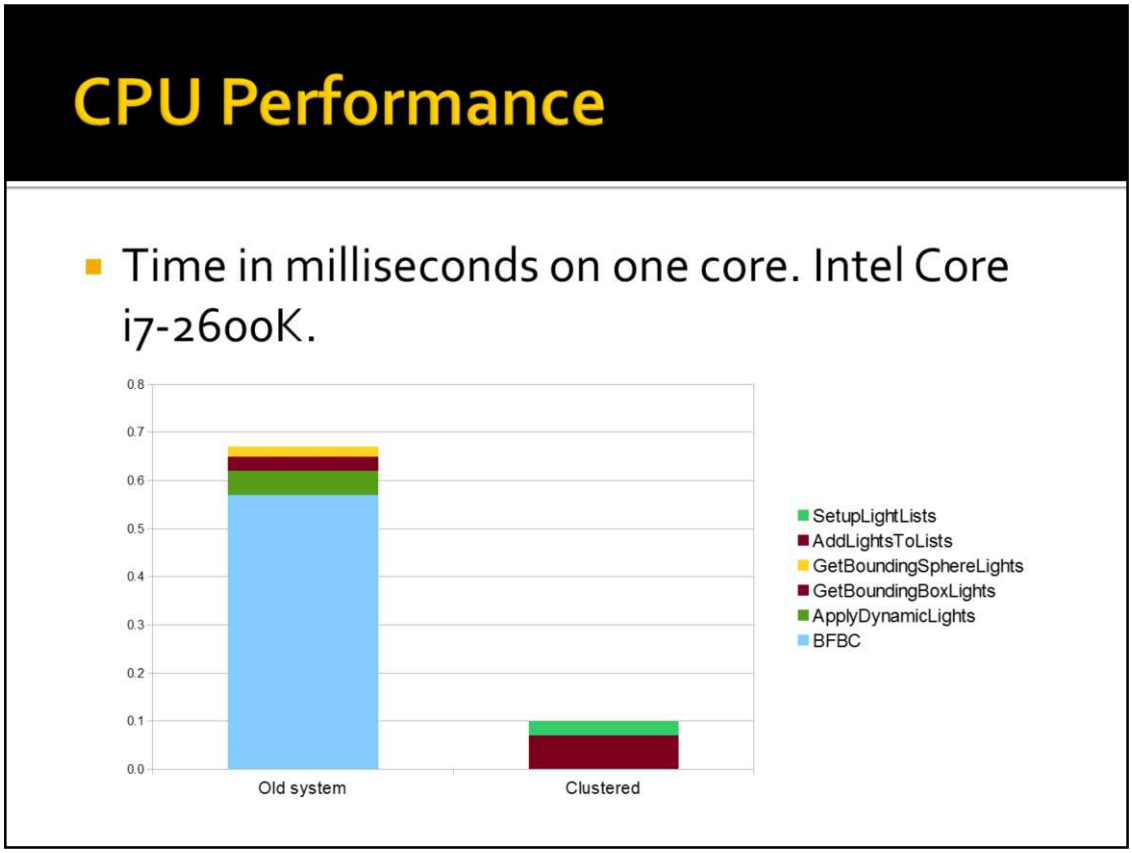
An observation that was made is that artists often place very similar looking lights close to each other. In some cases it is to get a desired profile of a light, in which case the two lights may in fact be centered at the exact same point. But often it is motivated by the real world, such as two headlights on car. Some vehicles actually have ten or more lights, all pointing in the same general direction. Rendering ten shadow buffers for that may prove to be far too expensive.

Shadows

- Decouple light and shadow buffers
  - Similar lights can share shadow buffers
  - Useful for car lights etc.



Often it works just fine to share a single shadow buffer for these lights. While the shadow may be slightly off, this is usually not something that you will notice unless you are specifically looking for it. To make this work the shadow buffer is decoupled from lights and the light is assigned a shadow buffer and frustum from which to extract shadows. The shadow frustum has to be large enough to include all the different lights that uses it.

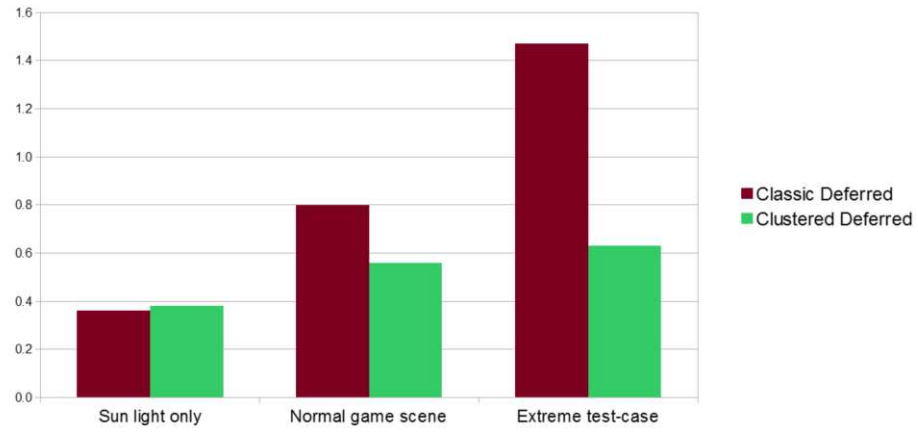


Given that we are doing the light assignment on the CPU, one may suspect that this will become a significant burden for the CPU. However, our implementation is fast enough to actually save us a bunch of CPU time over our previous solution. In a normal artist lit scene we recorded 0.1ms on one core for clustered shading. The old code supporting our previous forward pass for transparency that was still running in our system was still consuming 0.67ms for the same scene, a cost that we can now eliminate.

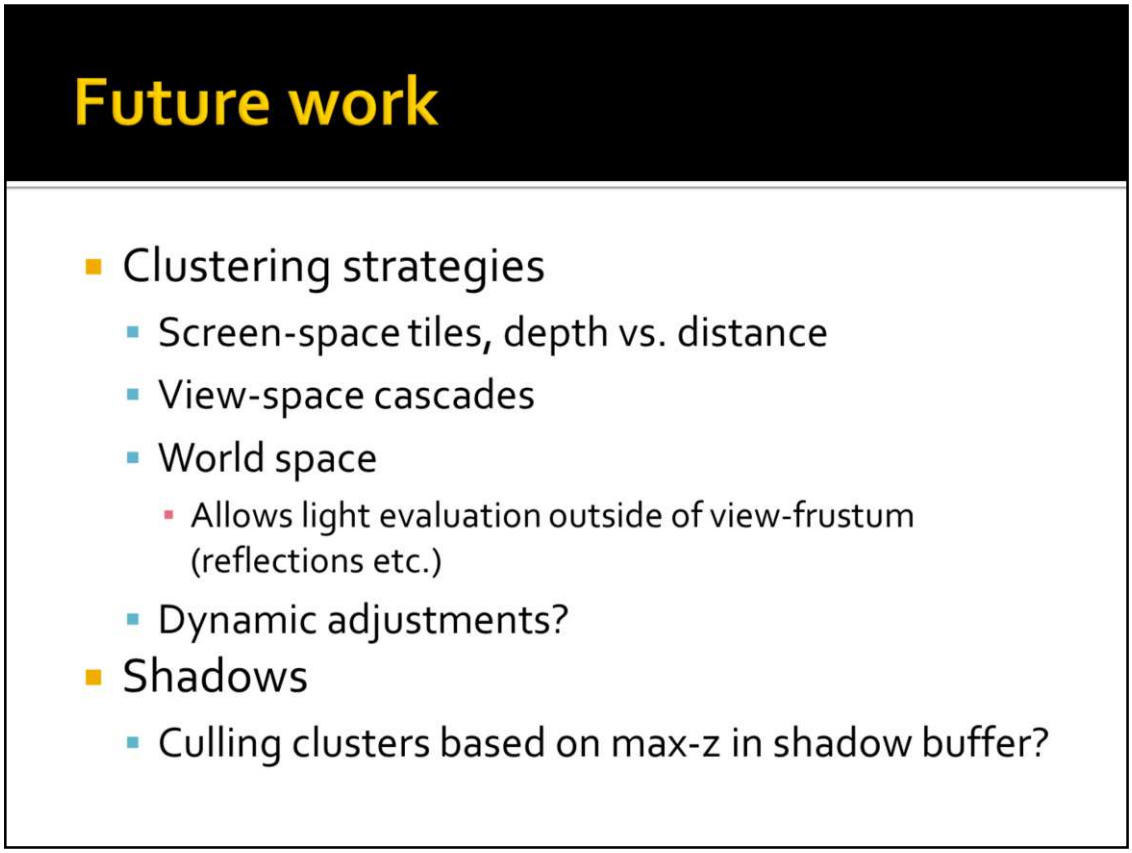
As of this writing, further optimizations have been made resulting in another 20-30% lower CPU cost than previously.

GPU Performance

- Time in milliseconds. Radeon HD 7970.



When we have nothing but the sun light in our scene, we incur a small overhead compared to classic deferred shading from looking up our empty light list and looping zero times. Once a light or two has been entered into the scene clustered shading is typically faster, and in regular artist lit scenes significantly so. Once we go to extreme artificial test cases with hundreds of lights sprinkled randomly in front of the player, clustered scales really well whereas classic deferred gets significantly slower. We have observed cases as large as 5x more expensive, whereas typically for heavy scenes it's around 2x. The difference is generally about how slow we can make classic deferred, rather than how fast clustered can be, as the clustered performance stays quite consistent, whereas classic's performance can very quite a lot depending on the scene.



We did some prototyping on a distance based clustering strategy instead of depth. While this allowed pointlights to be culled efficiently and exactly, this also made the cluster lookup slightly more expensive. The performance gain from an exact test was small enough that only with extreme workloads did we gain back what we lost from slower cluster lookup and we were hard-pressed to find a case where it ended up being faster in practice.

Another possible approach is clustering on view-space cascades. This would allow for exact AABB tests. One could argue that if you are going to test using an AABB, then you might just as well shape your clusters that way.

World-space clusters is another interesting option. While this would utilize the available clusters worse, the light distribution might match real world better. The other advantage is that you could evaluate light outside of the view frustum. This would allow for instance a reflection pass (such as a rear-view mirror) to use the same lighting structure for light evaluation.

There may be performance gains to be had if we consider the actual lights we have when clustering. For instance, we could tighten the cluster bounds if the most distant light active is closer than 500m, and the closest one more distant than 5m. This would allow for better cluster utilization.

Finally, a quick conservative reduction of the depth values in a shadow buffer could allow us to cull some clusters based on a conservative maximum-z value over some range. Whether this would result in any actual performance gains is unclear though.

Conclusions

- Clustered shading is practical for games
  - It's fast
  - It's flexible
  - It's simple
  - It opens up new opportunities
    - Evaluate light anywhere
    - Ray-trace your volumetric fog

What are you waiting for? Start writing your clustered shader today! ☺

References

- [Andersson 11] DirectX 11 Rendering in Battlefield 3. http://dice.se/publications/directx-11-rendering-in-battlefield-3/
- [Harada et. al. 12] Forward+: Bringing Deferred Lighting to the Next Level. https://amd.box.com/s/9g70td498gmxz5lq8py5
- [Harada 12] A 2.5D Culling for Forward+. https://sites.google.com/site/takahiroharada/storage/2012SA_2.5DCulling.pdf
- [Intel 14a] Forward Clustered Shading. https://software.intel.com/en-us/articles/forward-clustered-shading
- [Intel 14b] Clustered Shading Android Sample. https://software.intel.com/en-us/blogs/2014/07/30/clustered-shading-android-sample
- [Olsson et. al. 11] Tiled Shading. http://www.cse.chalmers.se/~olaolss/main_frame.php?contents=publication&id=tiled_shading
- [Olsson et. al. 12] Clustered Deferred and Forward Shading. http://www.cse.chalmers.se/~olaolss/main_frame.php?contents=publication&id=clustered_shading
- [Persson 10] Making it Large, Beautiful, Fast and Consistent: Lessons Learned Developing Just Cause 2. In *GPU Pro*. pp 571-596

Questions?

 @_Humus_

emil.persson@avalanchestudios.se

Part 3 – 40 Minutes
Presenter: Ola Olsson

Shadows for Many Lights

Shadows for many lights

- Paper from I3D 2014
  - "Efficient Virtual Shadow Maps for Many Lights".
- Builds on Clustered Shading
  - Efficient real-time many-light algorithm.
- Adds support for shadows
  - Hundreds in, real-time
    - Well, research-real-time at least.


Efficient Virtual Shadow Maps for Many Lights

Ola Olsson* Erik Sintorn* Viktor Kämpe* Markus Billeter* Ulf Assarsson*

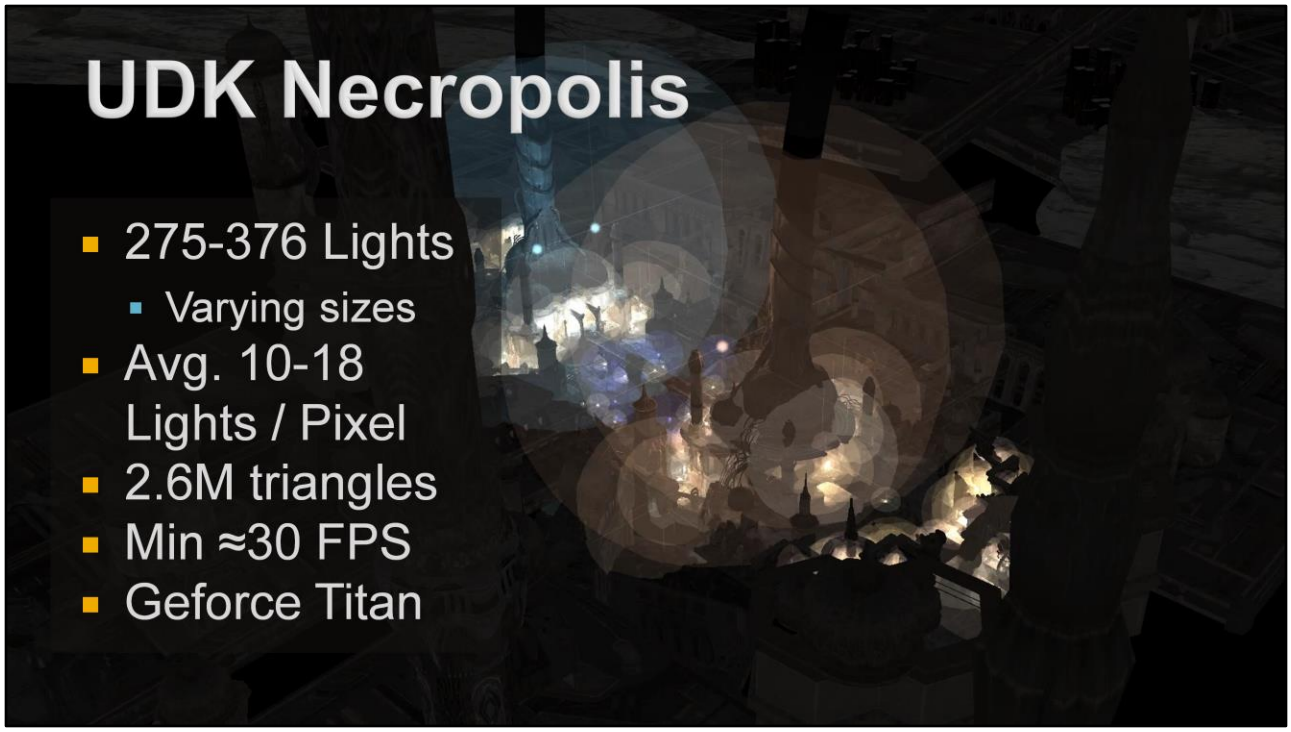
Chalmers University of Technology

Abstract

1 Introduction

1.1 Contributions

- This talk is about the techniques presented in my paper at I3D earlier this year…
- …titled Efficient Virtual Shadow Maps for Many Lights.
- This paper in turn builds on clustered shading,
- which is an efficient and robust real-time algorithm for many lights,
- and adds support for shadows from hundreds of **omnidirectional** lights in real time.

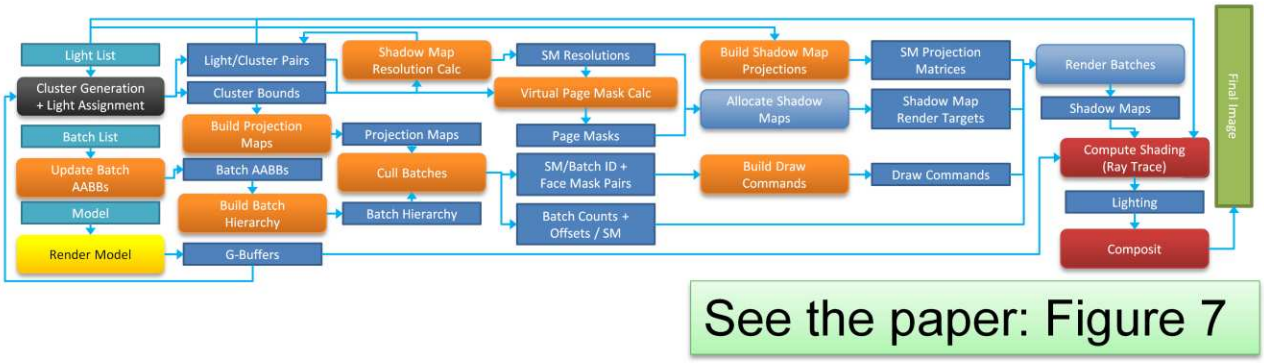


- To show what we achieve, here are some results from the paper
- This scene contains almost 400 shadow casting, omnidirectional, point lights,
- Some very large, most smaller.
- All lights and geometry is treated as dynamic.
- Even with almost 20 lights per pixel on average, we achieve…
- …a minimum of around 30 fps on a Titan GPU.

How?

Slide 5 / 500

- Well… Rather a few steps involved:



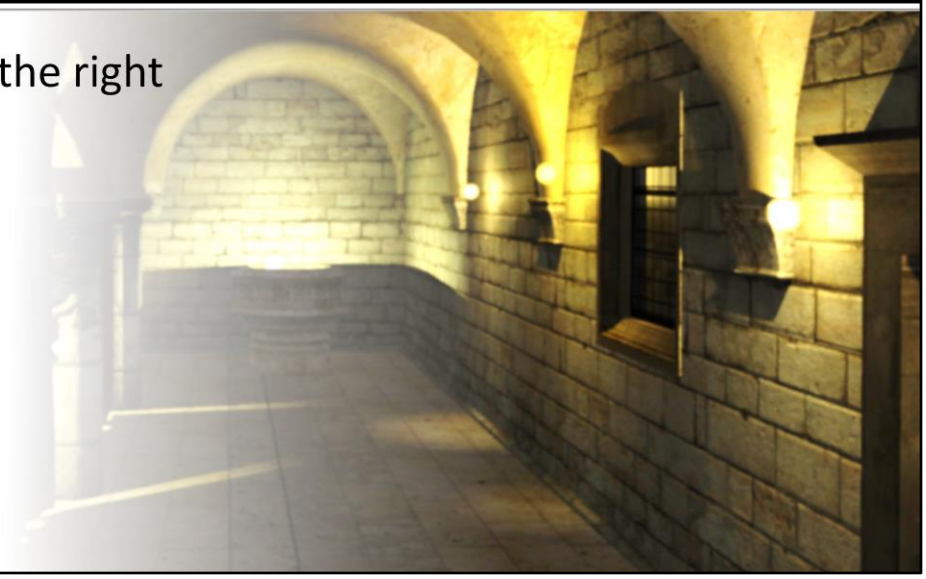
See the paper: Figure 7

- So how do we achieve this?
- This chart is in the paper, and shows the, rather many, stages and data flow in our system.
- In 20 minutes, I cannot hope to cover this in detail, so I will focus on a couple of important steps.

What's the problem?

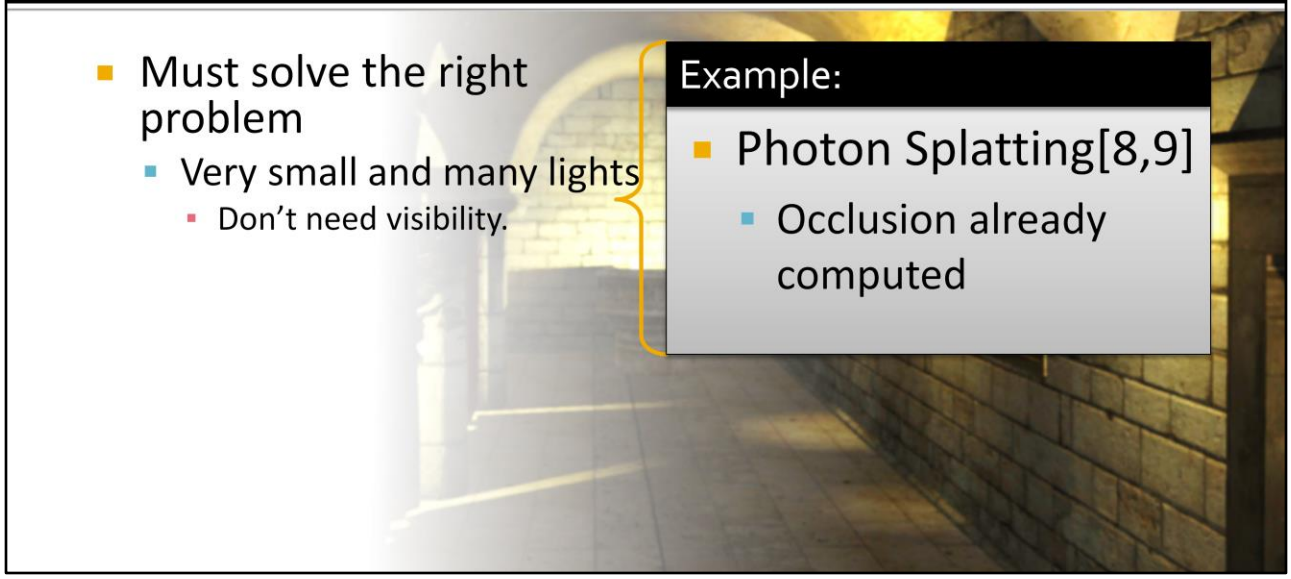
CHALMERS

- Must solve the right problem



- When approaching this problem it is necessary to limit the problem to be able to design an efficient solution.

What's the problem?



- On the one hand, very small and numerous lights, may not require visibility calculations at all.
- an example of this is photon splatting.

What's the problem?

CHALMERS

- Must solve the right problem
  - Very small and many lights
    - Don't need visibility.
  - Large lights with much overlap
    - Can approximate visibility.

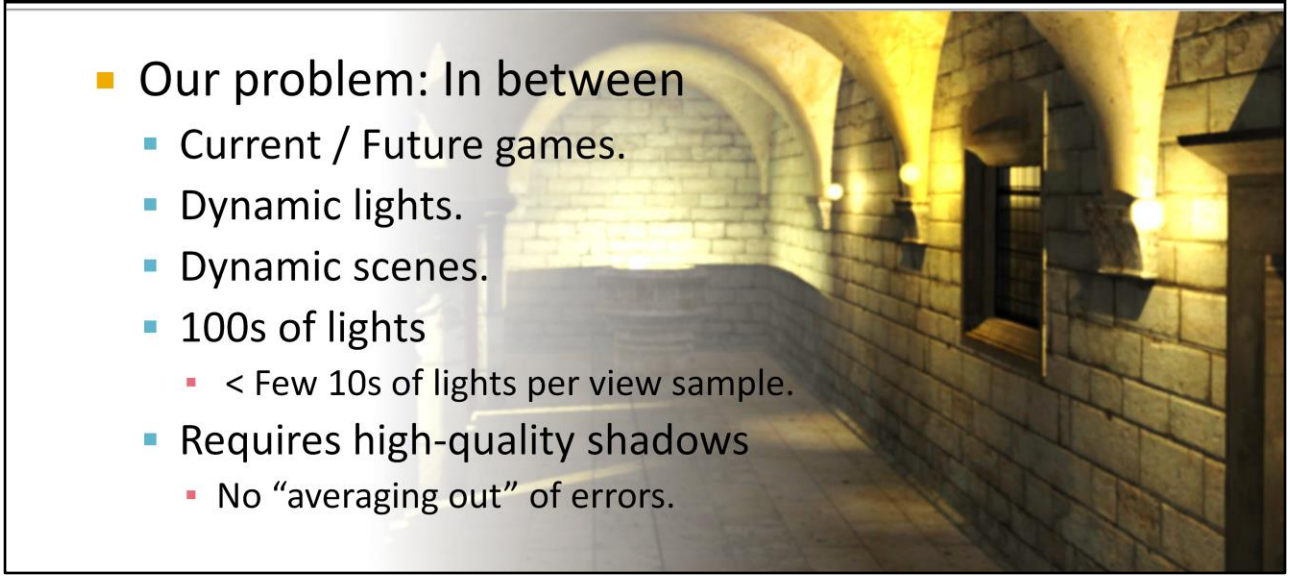
Example:

- Imperfect Shadow Maps[8]
- ManyLoDs[7]

- On the other hand, with many very large lights we can use approximate visibility
- As errors average out, as it is called.

What's the problem?

CHALMERS

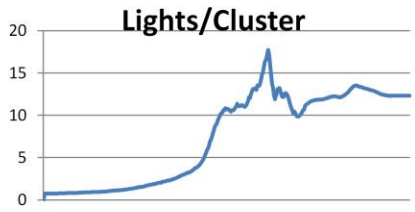


- In this paper, we aim for something in between these
- Targeting the numbers of lights and degree of overlap we might expect in modern games.
- We also target fully dynamic environments, so nothing is precomputed.
- The resulting numbers of lights per pixel implies that we must calculate high-quality shadows.
- As there will not be enough overlap to hide errors.
- I will now show the scenes we used in the paper to illustrate what we are trying to achieve

Light Distribution

CHALMERS

- UDK Necropolis
- 2.6M triangles
- 275-376 Lights
  - Varying sizes
- 10-18 Lights/Pixel

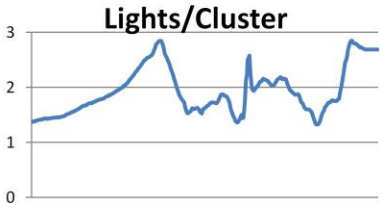


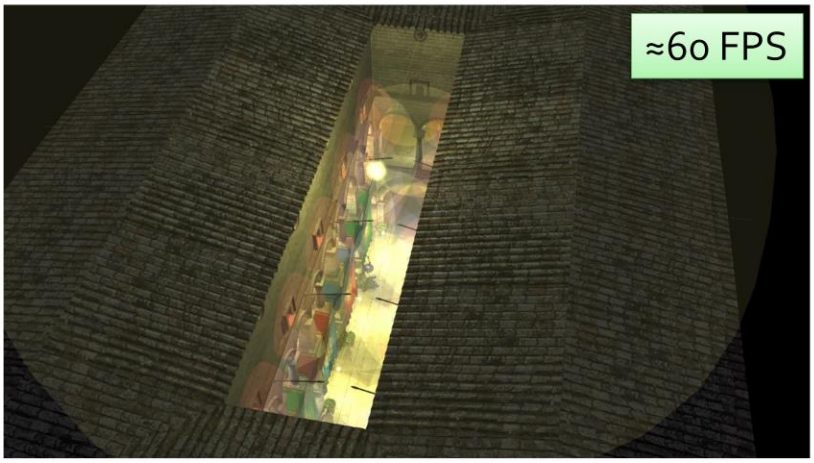
- In all the scenes, I've turned on drawing the geometry of the light spheres for illustration.
- In this scene some lights are very large, but the majority is smaller.
- This is the toughest scene we tested with 2.6M triangles,
- and we achieve a minimum of around 30 fps.

Light Distribution

CHALMERS

- Crytek Sponza
- 302K triangles
- 65 Lights
  - Varying sizes
- 1-3 Lights/Pixel



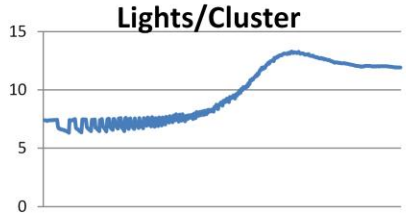


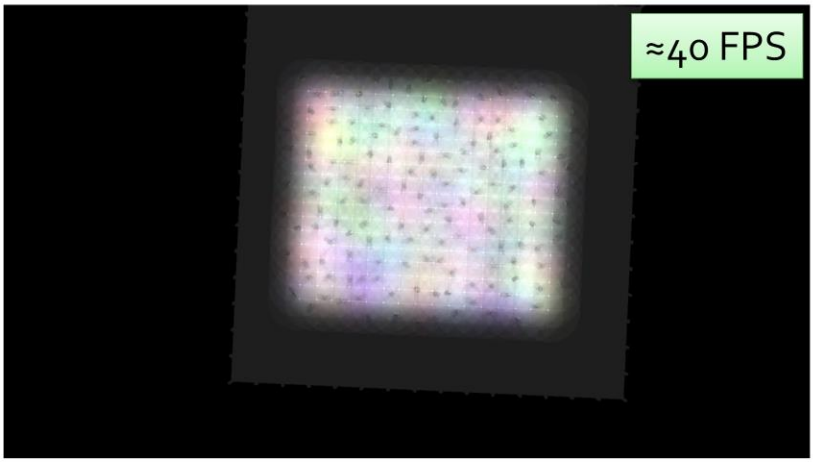
- In this simpler scene we achieve a minimum of roughly 60 FPS
- In this case there is on average less than 3 lights affecting each pixel (or equivalently, cluster)

Light Distribution

CHALMERS

- Houses
- 1M triangles
- 256 Lights
  - One size
- ≈13 Lights/Pixel



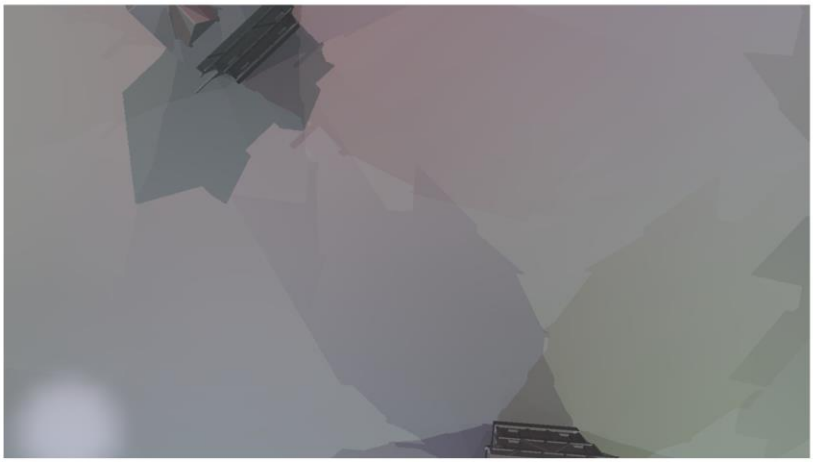


- This scene has very uniform distribution of lights, with about 12 lights per pixel.

Light Distribution

CHALMERS

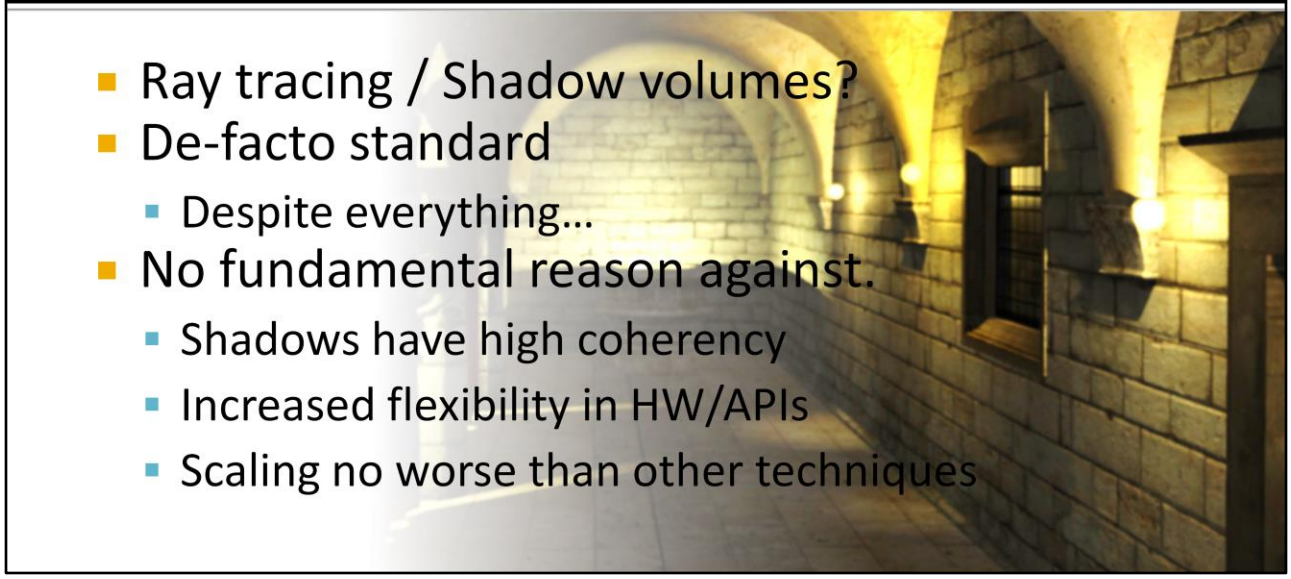
- Distinct shadows
- 12 Lights / Pixel



- Removing the lights spheres, leaving just the shadows.
- As you can see, each shadow cast is quite sharp, and so must be of a high quality.

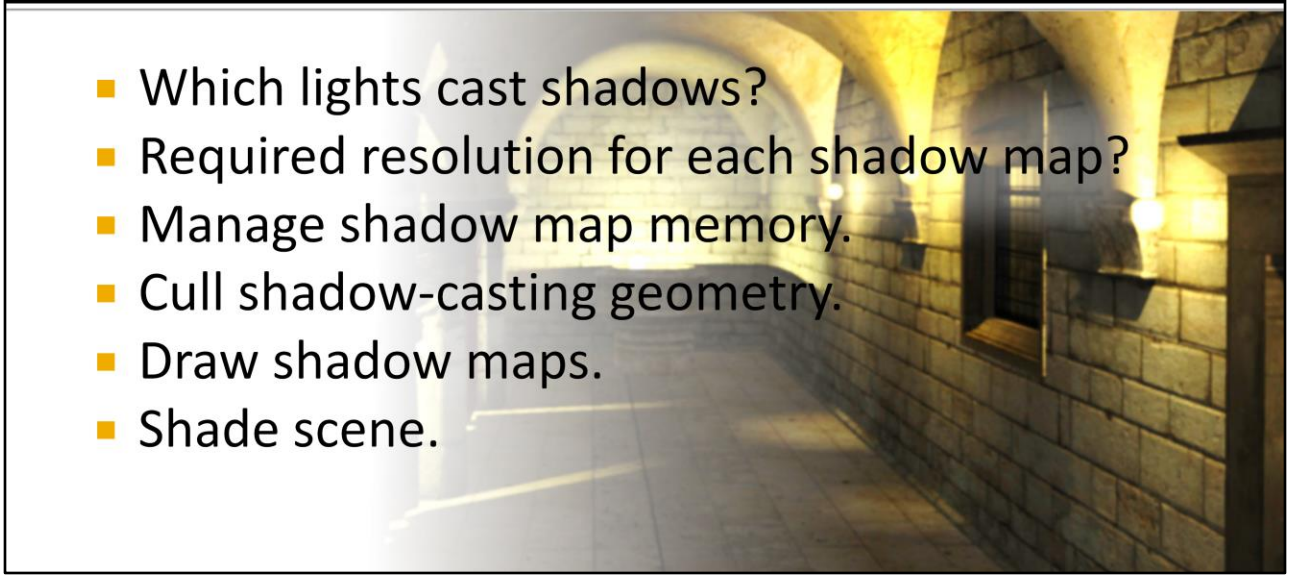
Why Shadow Maps?

CHALMERS



- Now some might wonder why we went with shadow maps?
- Although I'm guessing a lot of you are not wondering :)
- When I started this project, I had lots of more exciting ideas for how to do this.
- and it has even been stated that shadows maps are unsuited for many lights.
- However, when limiting the problem as I have just done, there really is no fundamental reason against shadow maps.
- And given that they are the de facto standard in the real-time industry, they must clearly be the first stop,
- if nothing else to provide a benchmark for more clever ideas.

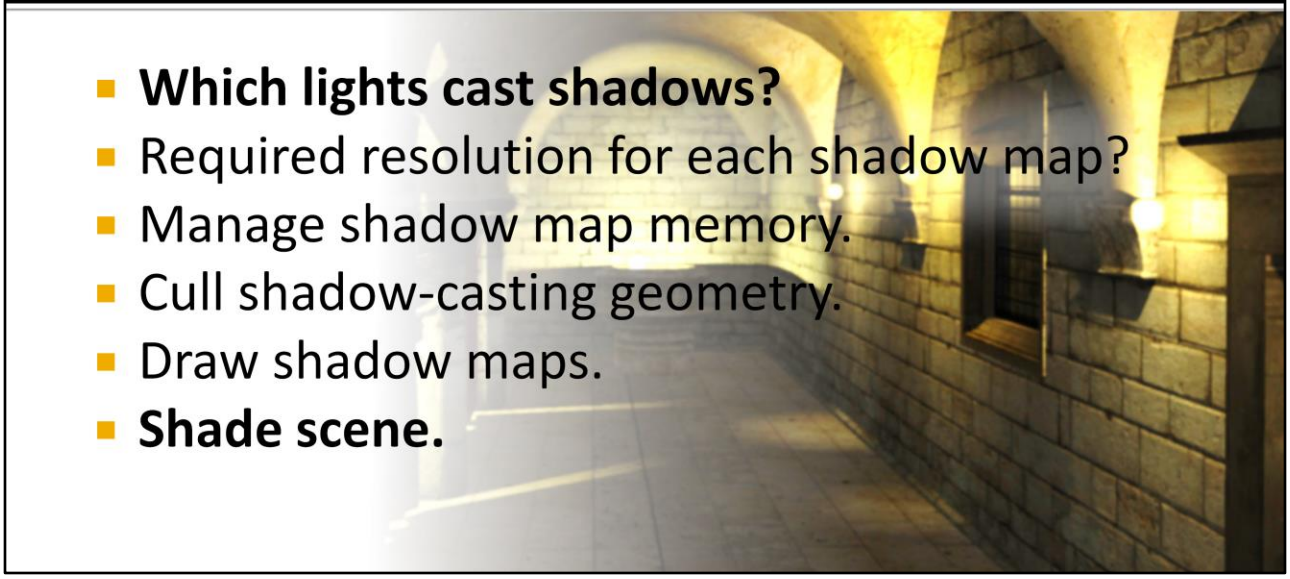
Problem breakdown



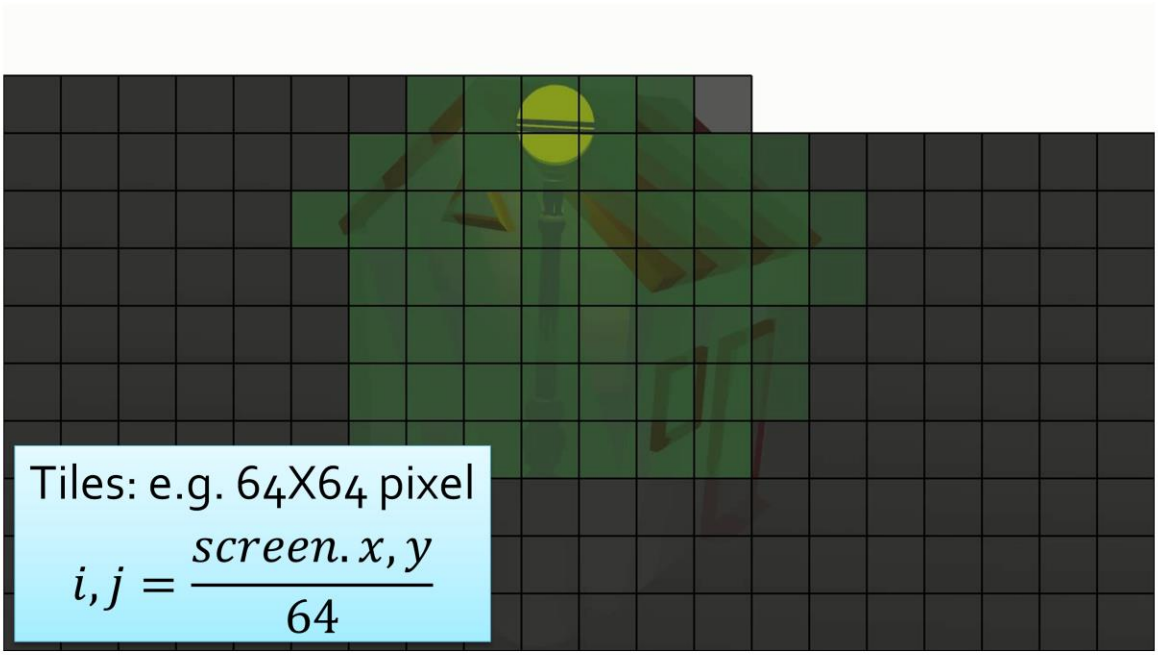
- Now, to create shadow maps,
- we need to perform the following steps each frame, using the current camera view.

Problem breakdown

CHALMERS



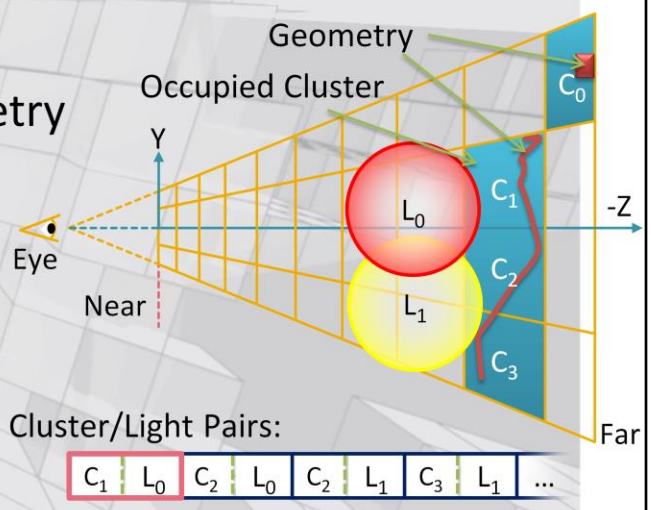
- The first step is the same as determining what lights are needed for shading
- And we have already seen how this can be achieved using clustered shading and other methods.
- The last step is also fairly trivial, using bindless textures, array textures or shadow map atlases.



- We solve this by using Clustered shading as the starting point of our algorithm.
- Recall that clustered shading is very efficient, coming very close to the minimal set of lights for shading,
- and so provides a good starting point for adding shadows.
- To recap, in its simplest form, it is the extension of screen space tiles, shown here…(click)
- into 3D, by placing regular subdivisions along the depth direction.

Clustered Shading Key Features

- Clusters
  - Proxy for visible geometry
    - Bounds view samples
      - 3D boxes
    - Orders of magnitudes fewer
  - Cluster/Light Pairs
    - Links clusters and lights



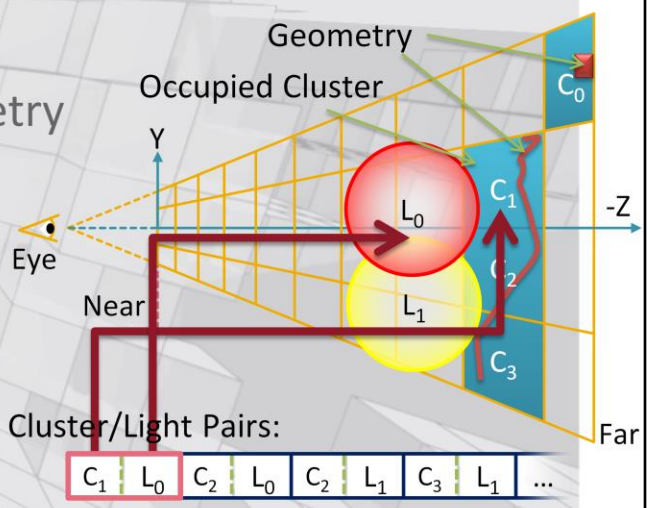
Cluster/Light Pairs:

C_1	L_0	C_2	L_0	C_2	L_1	C_3	L_1	...

- Note that the clustered shading method we build upon here is the one described in the paper, not Emils eminently practical method.
- The reason for this is that we need some information about the shadow receiving geometry to be able to get good shadow performance. Therefore, we cannot just use the up-front full grid approach.
- Two things about clusters are worth repeating in this context:
- First, the visible geometry is approximated reasonably well by the clusters, but there are very few in comparison. Usually around a thousand times more pixels than clusters, though this is of course tuneable with cluster size.
- Secondly, clustered shading provides an association between clusters and lights. For shading it allows us to know which lights
- affect each cluster, but also the reverse mapping, which we will make heavy use of to compute shadows.

Clustered Shading Key Features

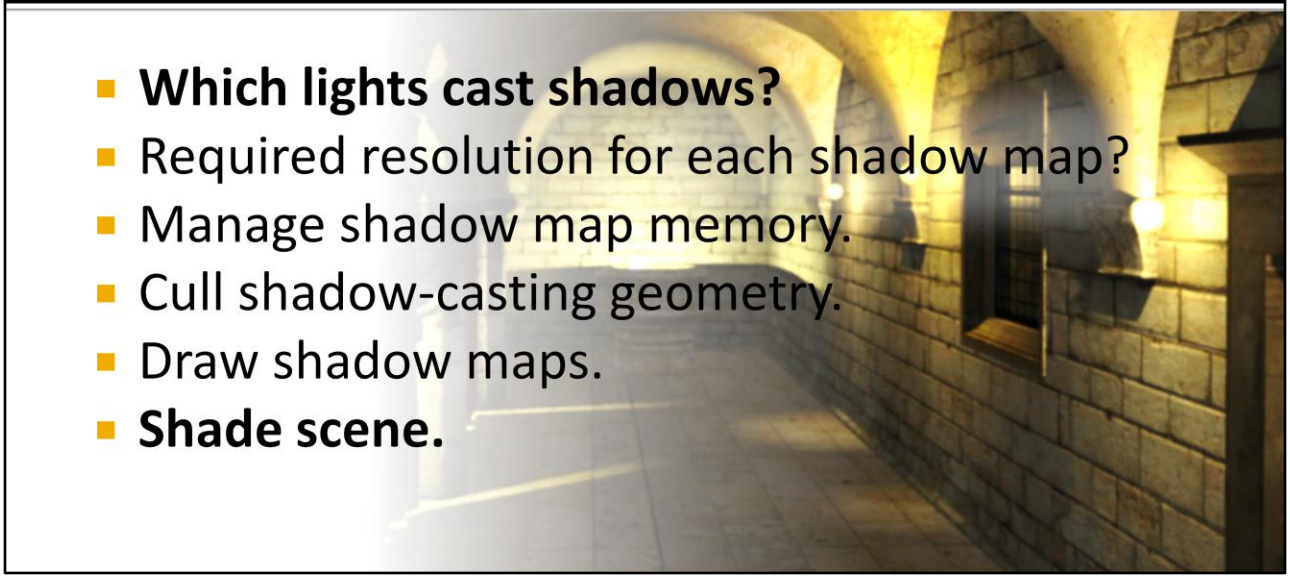
- Clusters
  - Proxy for visible geometry
    - Bounds view samples
      - 3D boxes
    - Orders of magnitudes fewer
  - Cluster/Light Pairs
    - Links clusters and lights



- The key data is the cluster/light pairs that enable parallel operations on these pairs.
- We use this for a lot of the processing later on.
- In the illustration, the pair of integers link the L0 light and the C1 cluster, the cluster is shown here as containing geometry, and overlapping a light, and it is these that we will process.

Problem breakdown

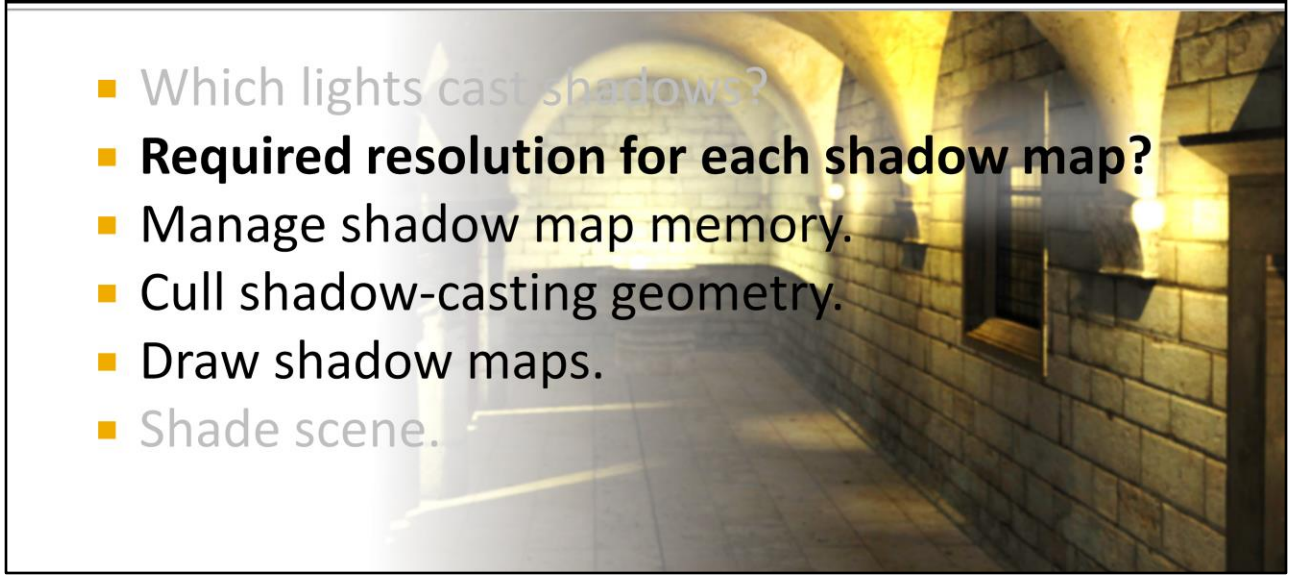
CHALMERS



- So, the clustered shading algorithm provides the lights affecting each cluster, and a light that does not affect any cluster need not be processed further.

Problem breakdown

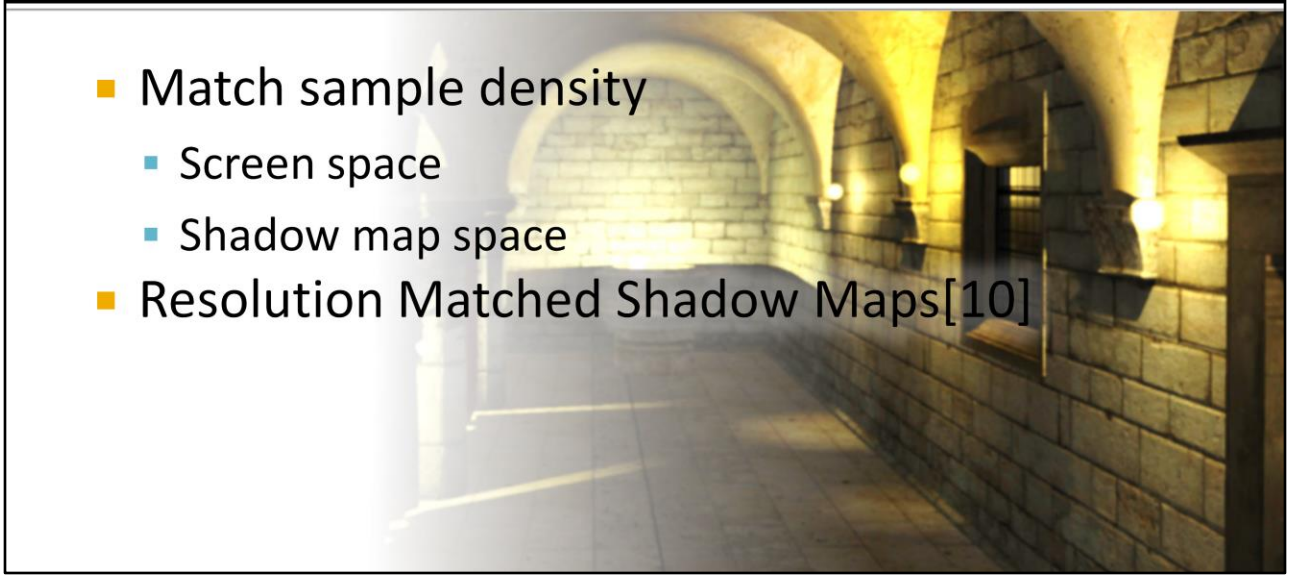
CHALMERS



- Next we must work out the resolution of each shadow map
- We could of course just pick a constant resolution…
- but then we'd not be talking about high quality shadows any more, with over and undersampling both being the norm.
- Using oversampled shadow maps is not just wasteful in terms of memory, it is also slow and may yield aliasing.

Shadow Map Resolution

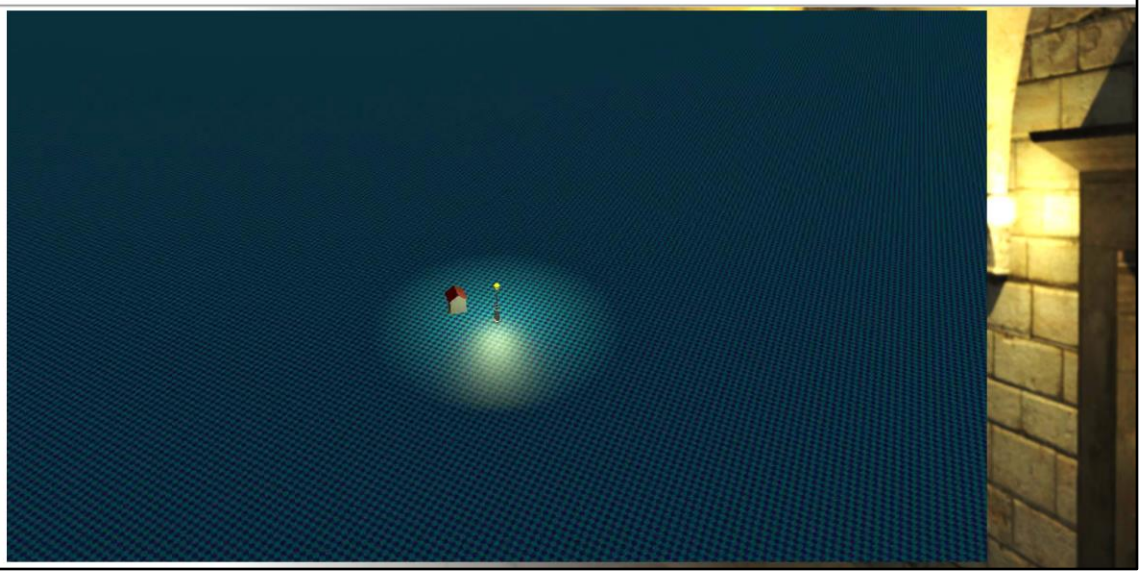
CHALMERS



- So what we need to do is somehow match the resolution of the shadow map to the
- Density of the projection of the visible samples.
- As done in the technique Resolution Matched Shadow Maps, we'll take this idea.

Example Scene

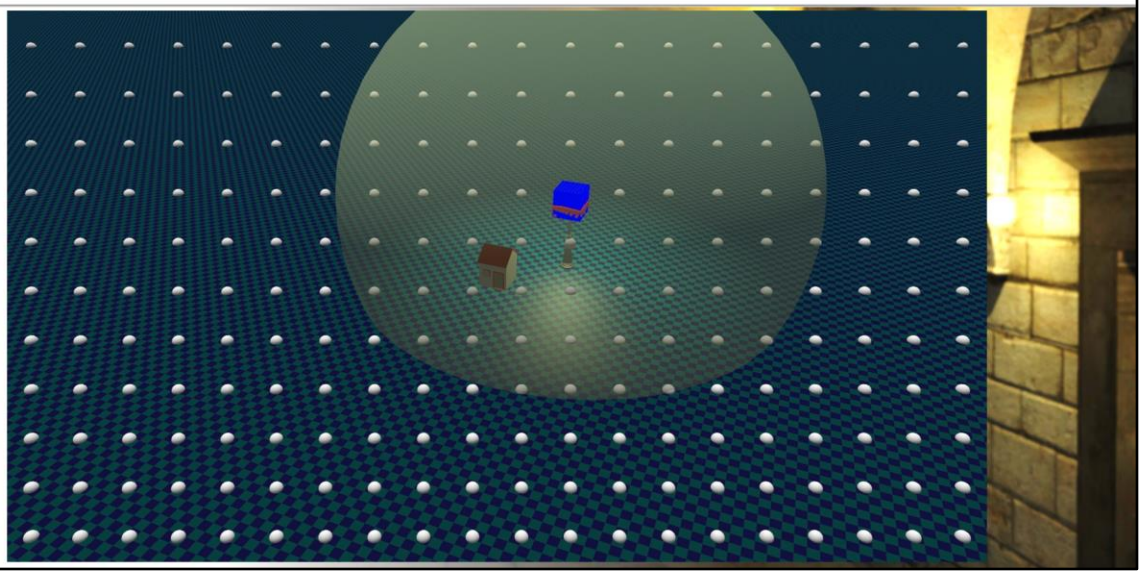
CHALMERS



- SO using this as the starting point I will try to illustrate the idea
- House, lights sphere, shadow cube map

Samples – Distant View

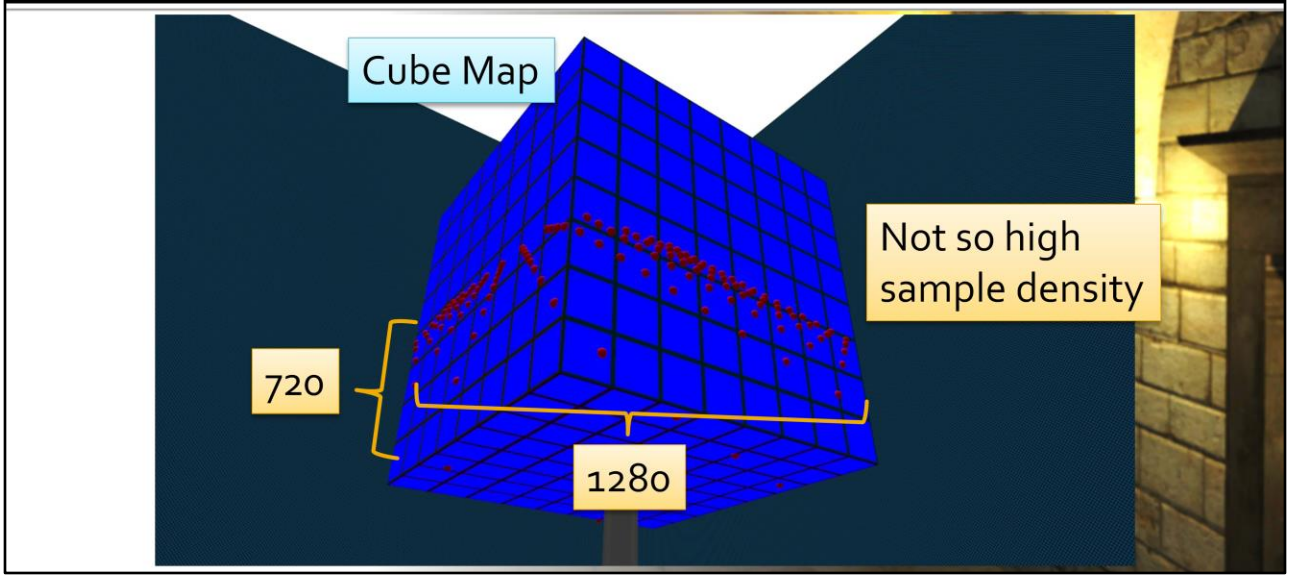
CHALMERS



- View sample representatives, from a relatively distant view

Projection on Shadow Map

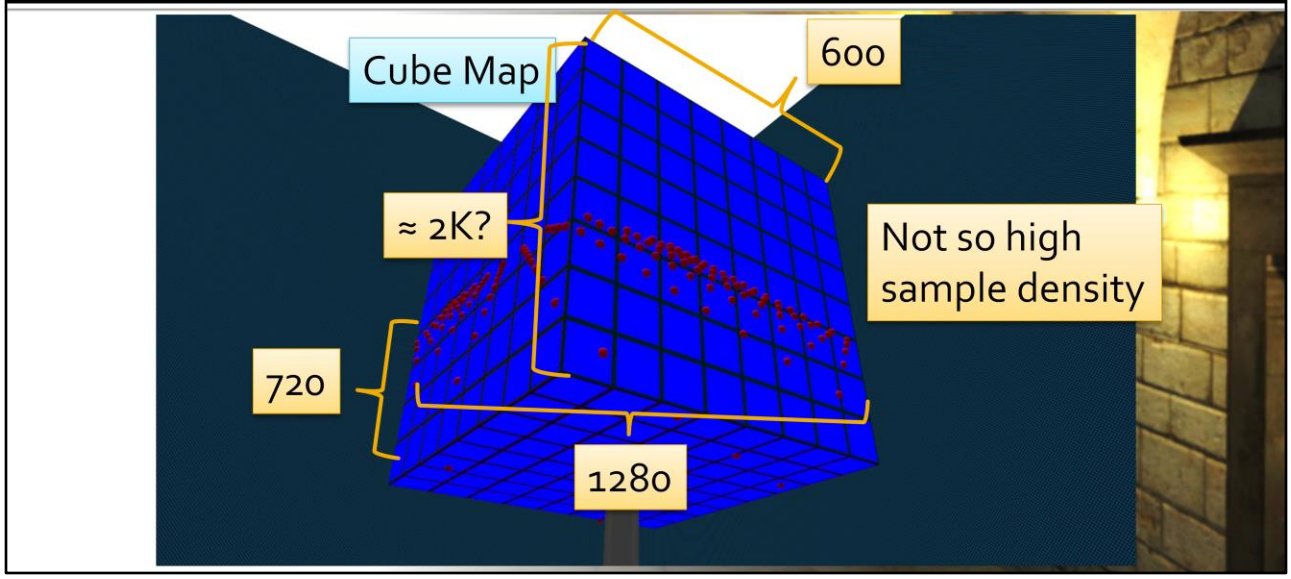
CHALMERS



- And their projeciton on the cube map.,
- The density, assuming this was a standard definition render, is not extreme…

Projection on Shadow Map

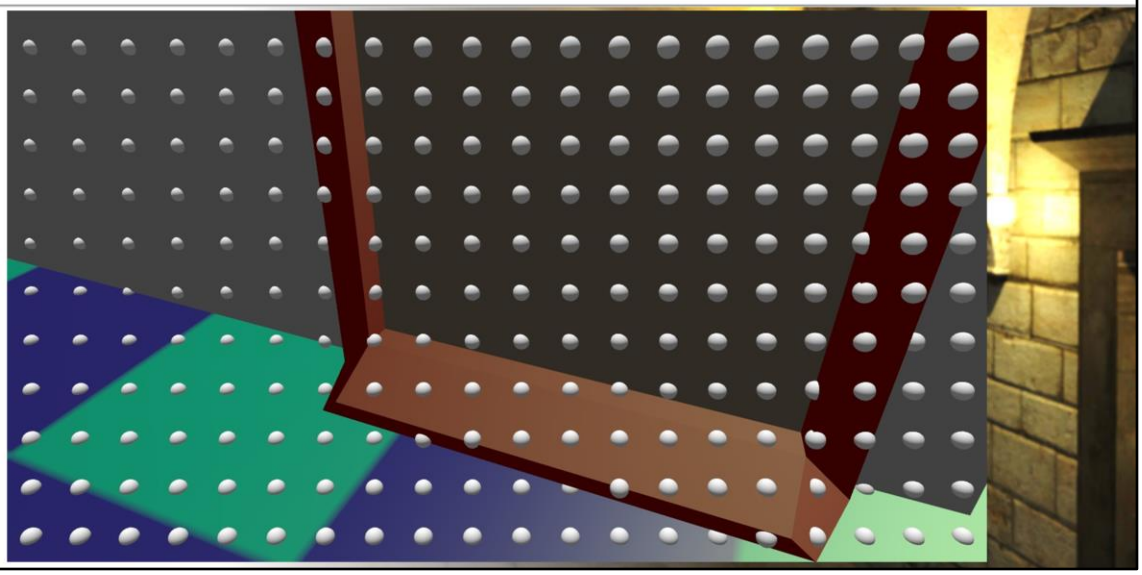
CHALMERS



- with this sample distribution, the shadow map might need a 2k by 600 resolution.

Samples

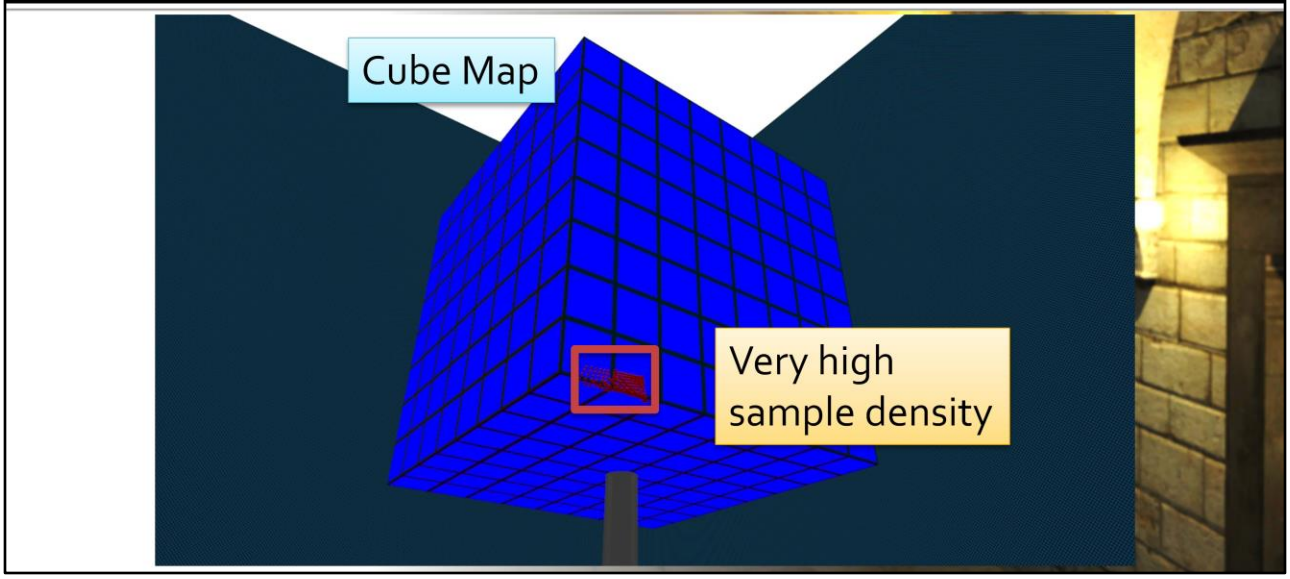
CHALMERS



- With this view which is zoomed in on a small part of the house,
- And thus concentrating the samples in world space…

Projection on Shadow Map

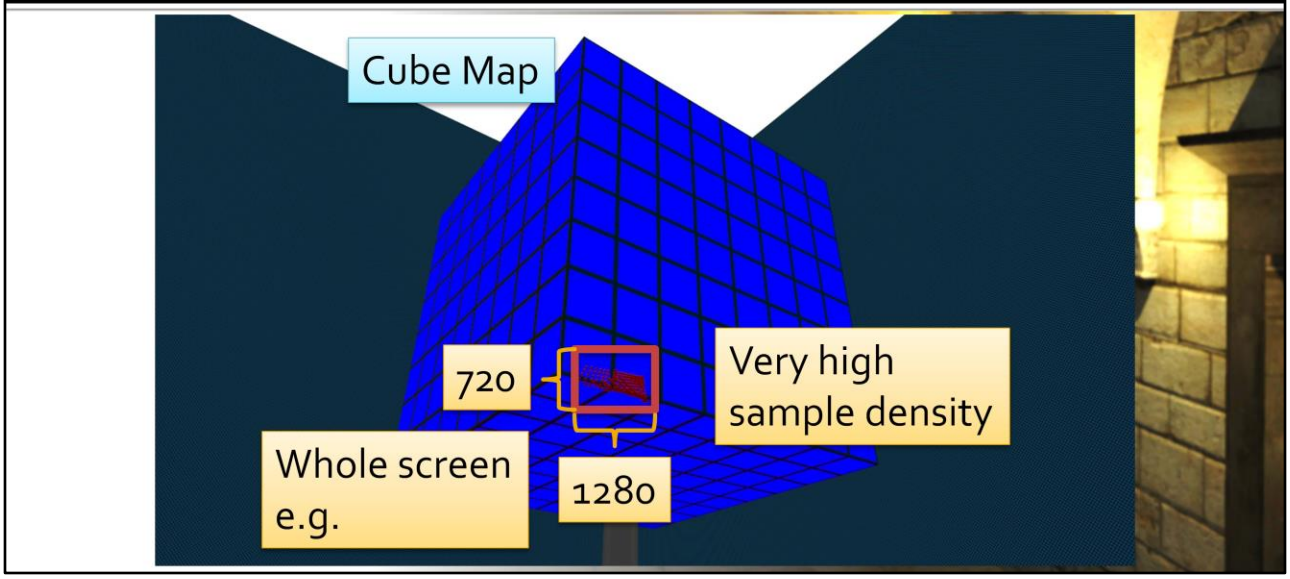
CHALMERS



- We get this very dense projection onto the cube map

Projection on Shadow Map

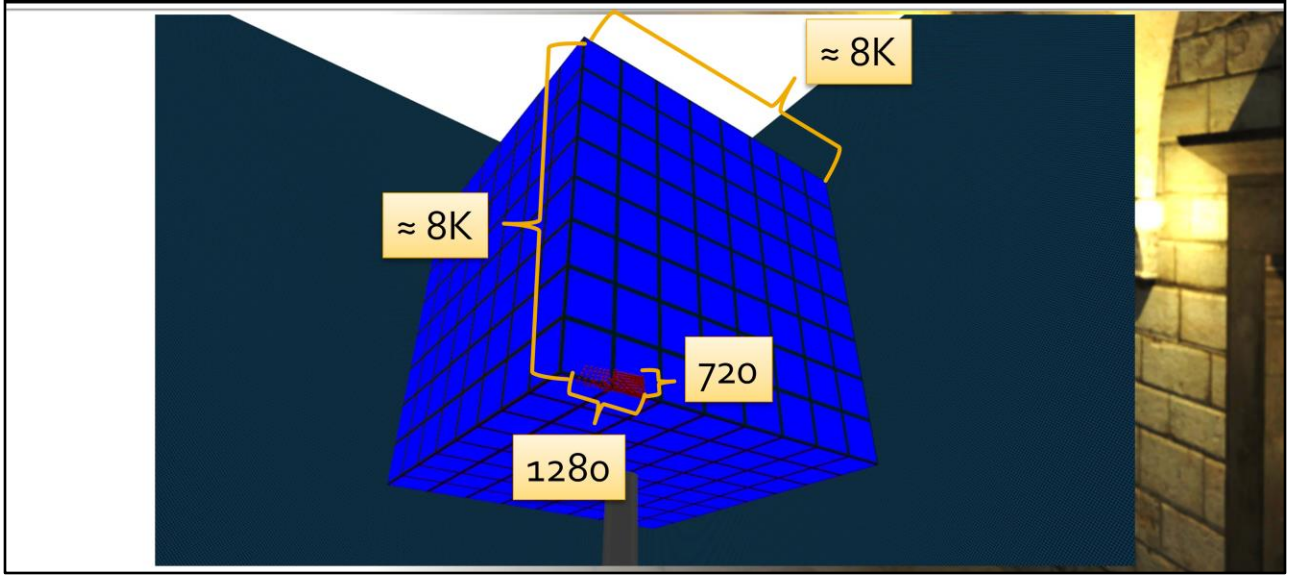
CHALMERS



- Again with a waving the hands calculation…

Projection on Shadow Map

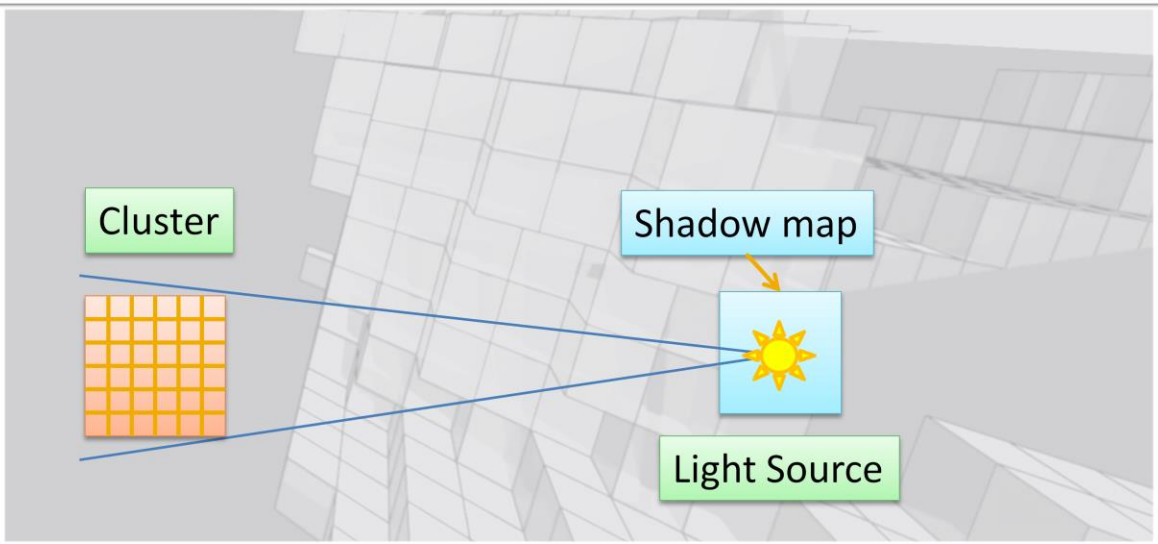
CHALMERS



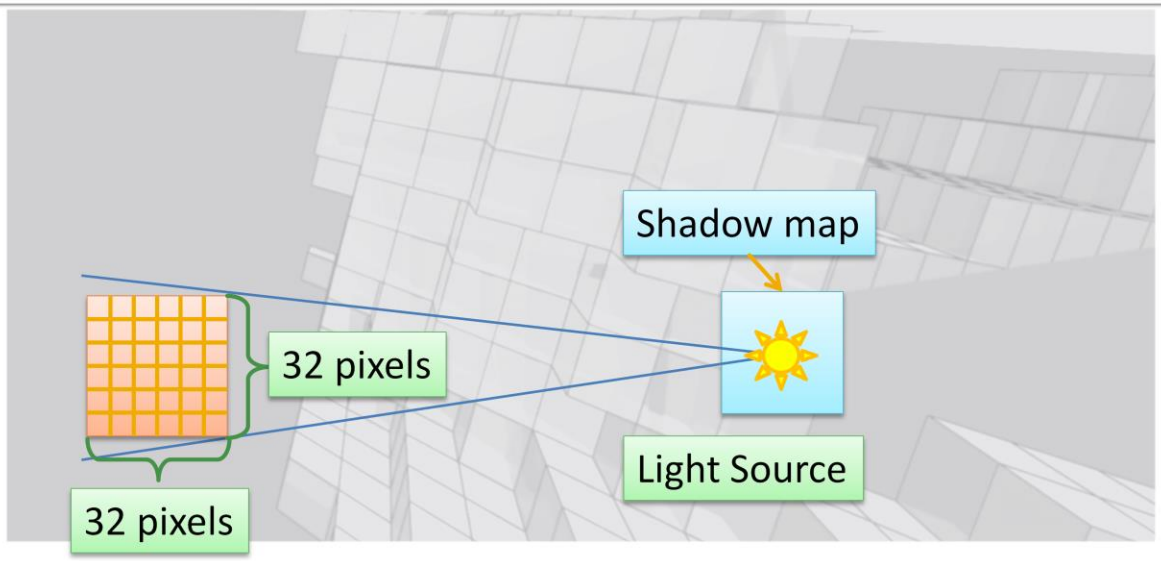
- Now, we could perhaps calculate the shadow map space derivatives for each sample individually, and use this to find the required resolution, by using the highest value.
- However, this can be quite expensive, and is

Clusters

CHALMERS

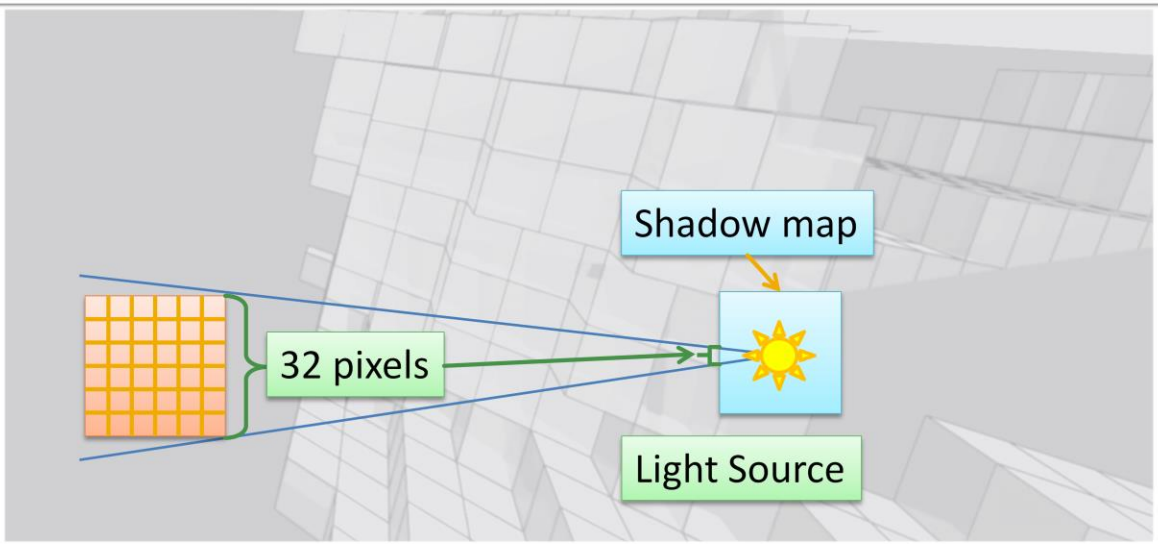


Clusters



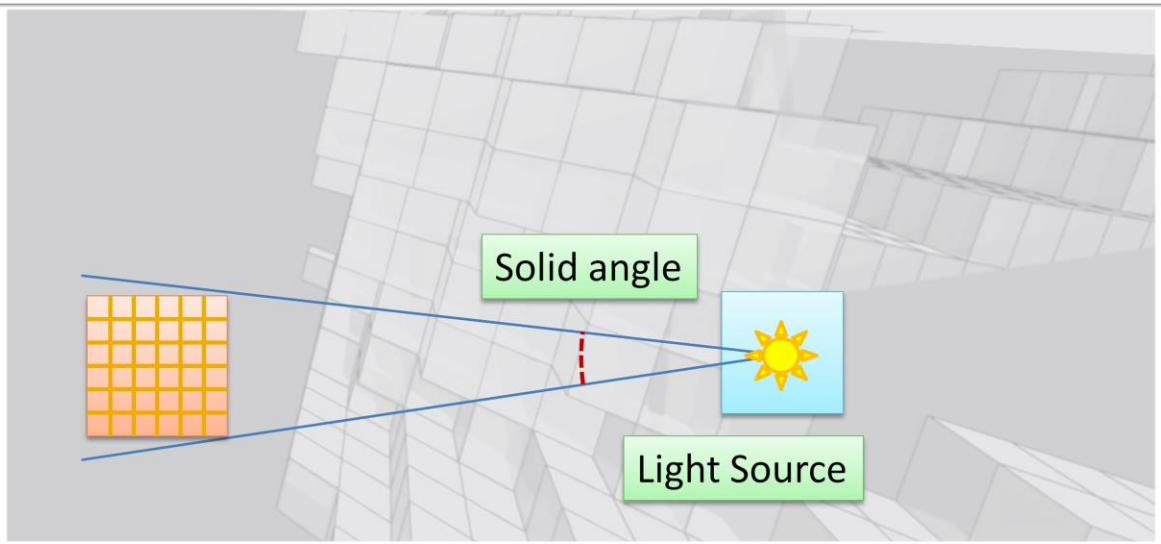
Clusters

CHALMERS



Clusters

CHALMERS

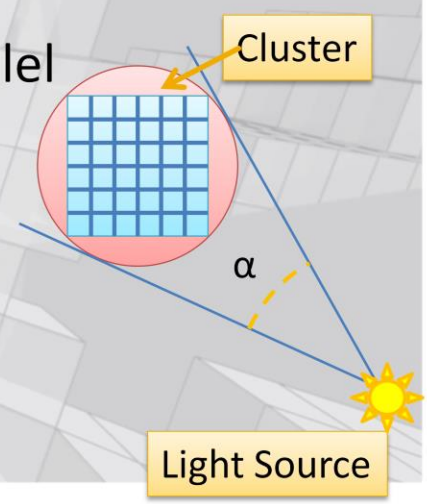


Clusters

Clusters

CHALMERS

- Solid angle over light sphere
- All cluster/light pairs in parallel
  - Keep highest for each light
    - `atomicMax`



Res calc.

CHALMERS

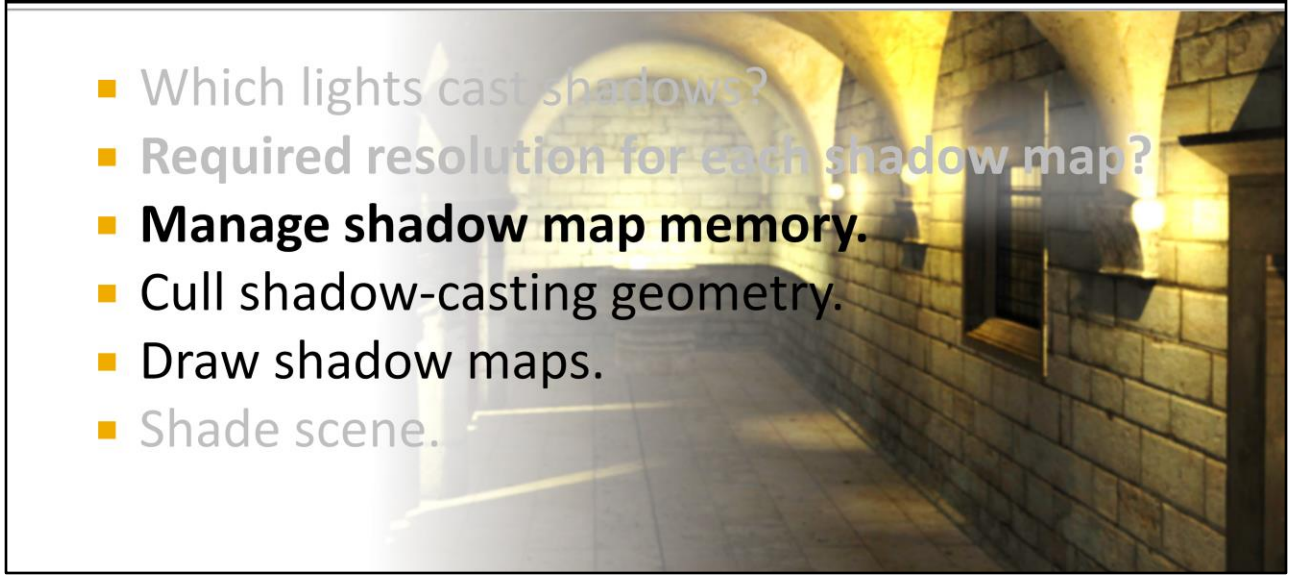
$$Res = \sqrt{\frac{S/(\alpha/4\pi)}{6}}$$

α = solid angle

S = Screen space foot print (e.g., 32x32)

Problem breakdown

CHALMERS



- To implement efficient and high-quality shadow maps for hundreds of lights.

Shadow Map Storage

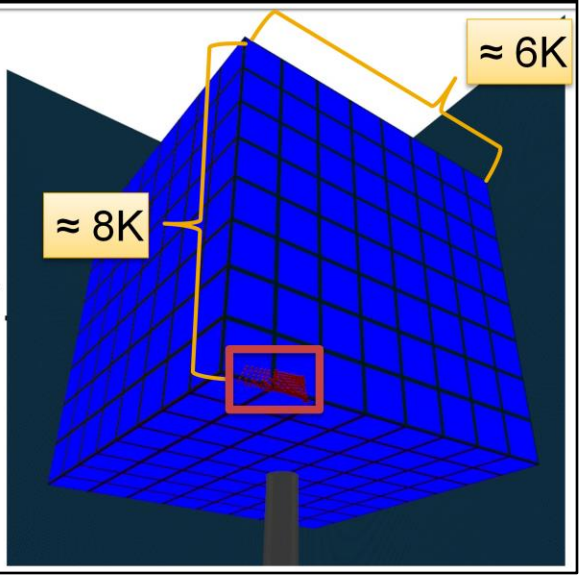
- Modern real-time shading algorithms
  - Tiled/Clustered Deferred/Forward*[1,2]
  - Inner loop in shader over ***light list***
- Shadow maps required up front.
  - Efficient ***storage*** paramount.
  - Ideally: Store no samples that are not used!
    - Somewhat tricky in practice.
    - (Shadow Volumes , Ray tracing, Alias-Free Shadow Maps)

*AMD: "Forward+" ⇔ Tiled Forward Shading

- Our solution is based on clustered shading, which is a modern real-time shading algorithm.
- These all have in common, apart from being adopted by most high end game engines…
- that the inner loop, in the fragment shader, iterates over a list of lights.
- This means all shadow maps must exist up-front before the shading pass.
- Consequently, efficiently managing shadow map storage has become a very important problem.
- And this does not just mean parceling out shadow maps as needed,…
- but ideally we should be able to store only those samples that matter.
- Several shadow algorithms do just that, but fail to achieve consistent real-time performance,
- For different reasons.

Shadow Map Memory

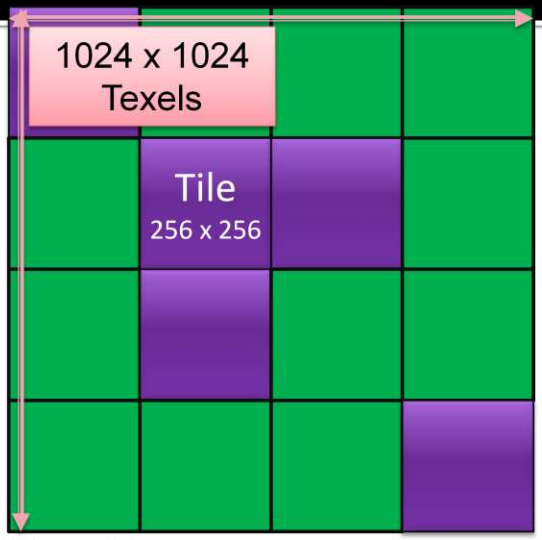
- 6 x 8K x 8K x 2 byte
  - 768Mb
- Only small part used!
  - Rest will not be touched



- As we saw before, the shadow map samples can be very tightly grouped, requiring a high shadow map resolution
- And note how this happens when most of the shadow map would be unused!
- This is pretty much how it has to be, as the high density comes from looking at something very near the camera,
- and then we're guaranteed to not see so very much of the scene.
- This is highly wasteful and a fantastic opportunity, for...

Hardware Virtual Textures*

- Allocate Virtual Texture Storage
  - Just like virtual memory.
  - Low overhead.
- Physical backing in Tiles (Pages)
  - E.g. 256 x 256 Texels.
- Recent APIs.
  - OpenGL 4.4 extension [4]
    - `ARB_sparse_texture`
  - DirectX 11.2
    - Tiled Resources [5]
  - Mantle? Consoles? (GCN)

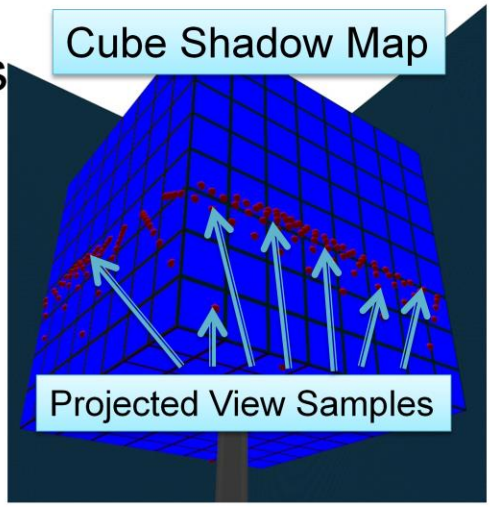


* Or: Sparse Textures, Partially Resident Textures

- For this reason we turn to hardware virtual, or sparse, textures
- These have become available in mainstream graphics APIs recently,
- and makes it possible to allocate large virtual textures,
- and commit physical storage in tiles only where needed.
- This fits our ideal pretty well, especially given that shadow map samples tend to clump together because of coherency

What to commit?

- Compute shadow lookups
  - For all view samples,
  - On all shadow maps.
- Commit touched pages.
  - Store a flag each.

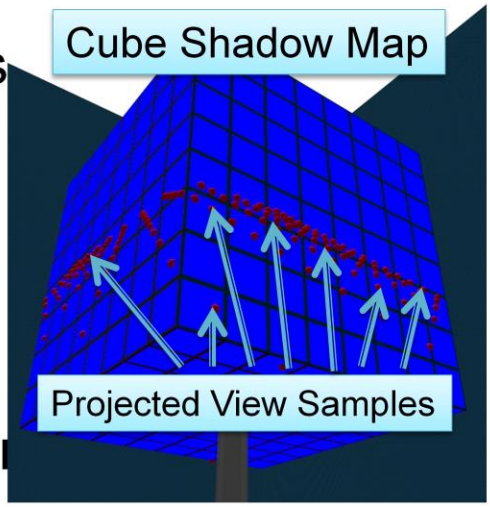


- So we must work out which physical tiles to commit.
- Essentially, we can solve this by performing all the shadow lookup once **before** drawing the shadow maps.
- and commit the touched pages.

TODO: Image/Illustration showing the pages that are committed (lift from other presentation)

What to commit?

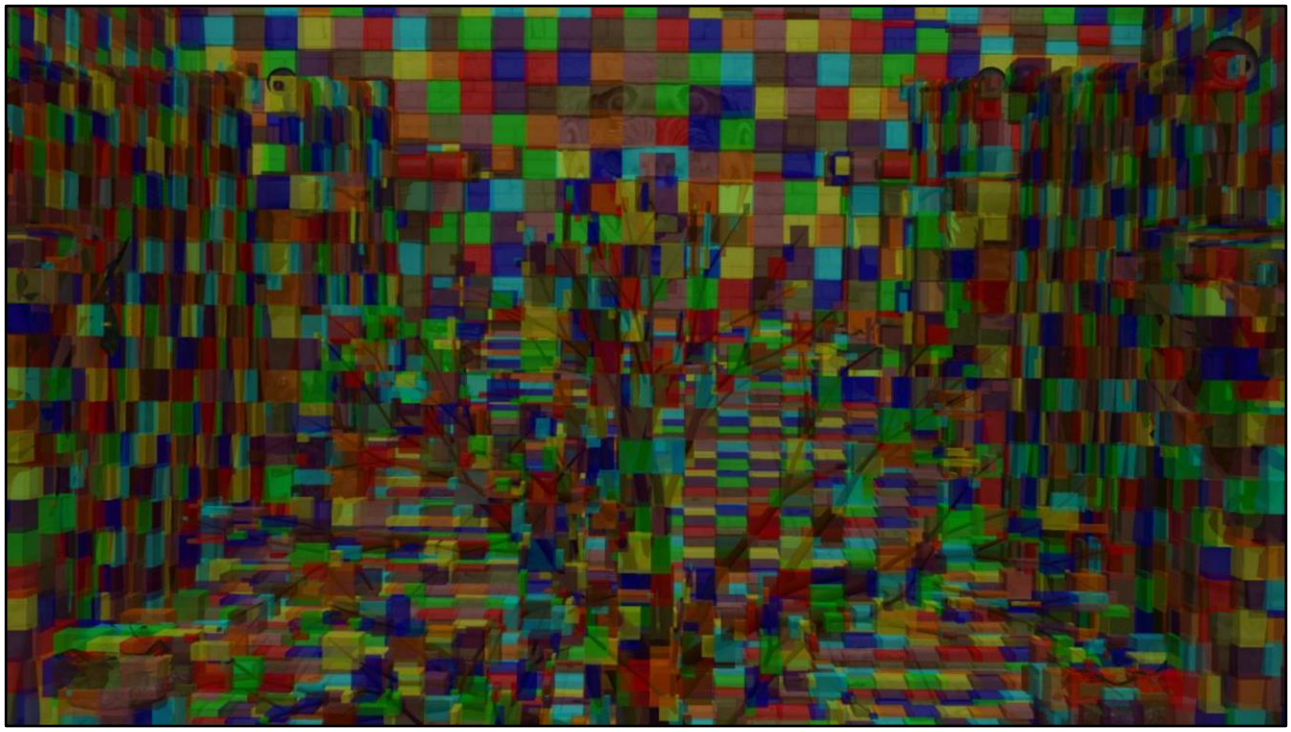
- Compute shadow lookups
  - For all view samples,
  - On all shadow maps.
- Easy, but…
  - Sounds expensive!
  - #samples x #lights
- (Re)Enter clustered shadi



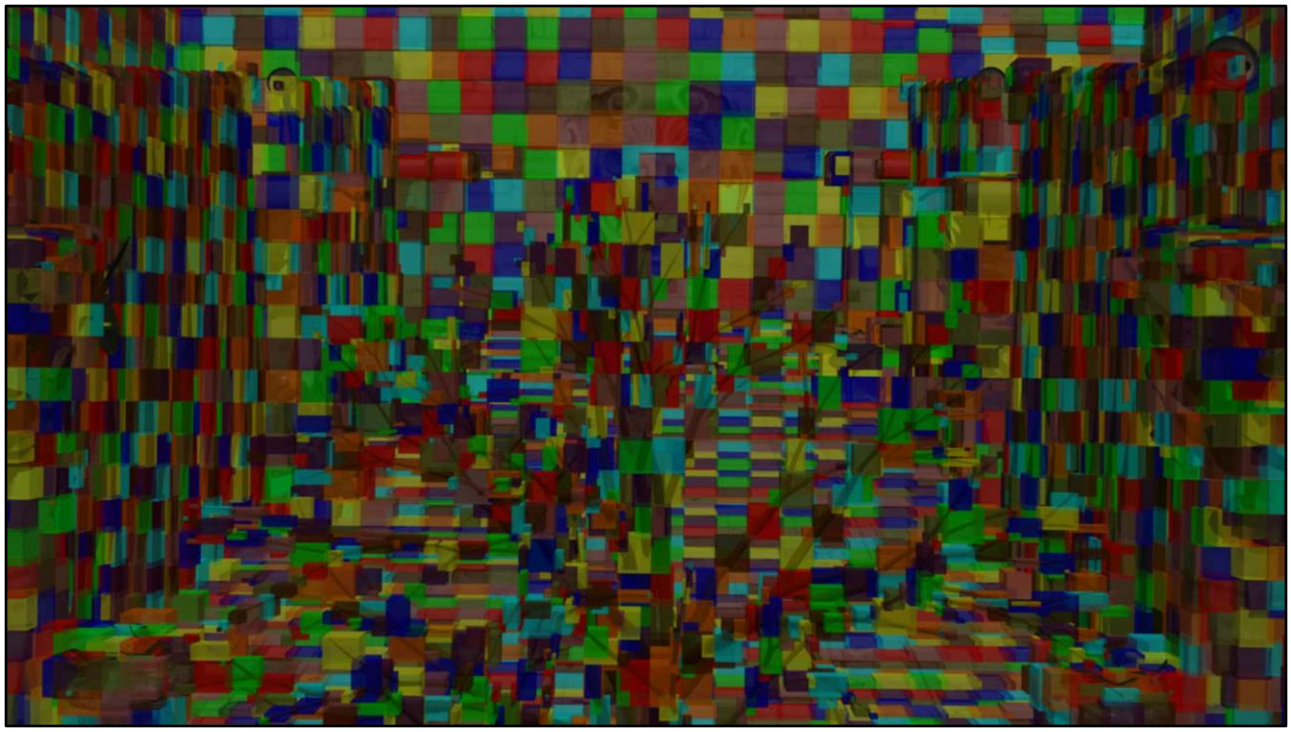
- So that is pretty simple in principle, but…
- If we have a couple of million pixels and a few hundred lights, this is going to be a bit inefficient!
- Using the cluster/light pairs shown earlier helps cut down both of these numbers significantly.
- We now only need to consider clusters, and only the relevant lights.

- Looking at this visually,
- Here is that modified crytek sponza again.

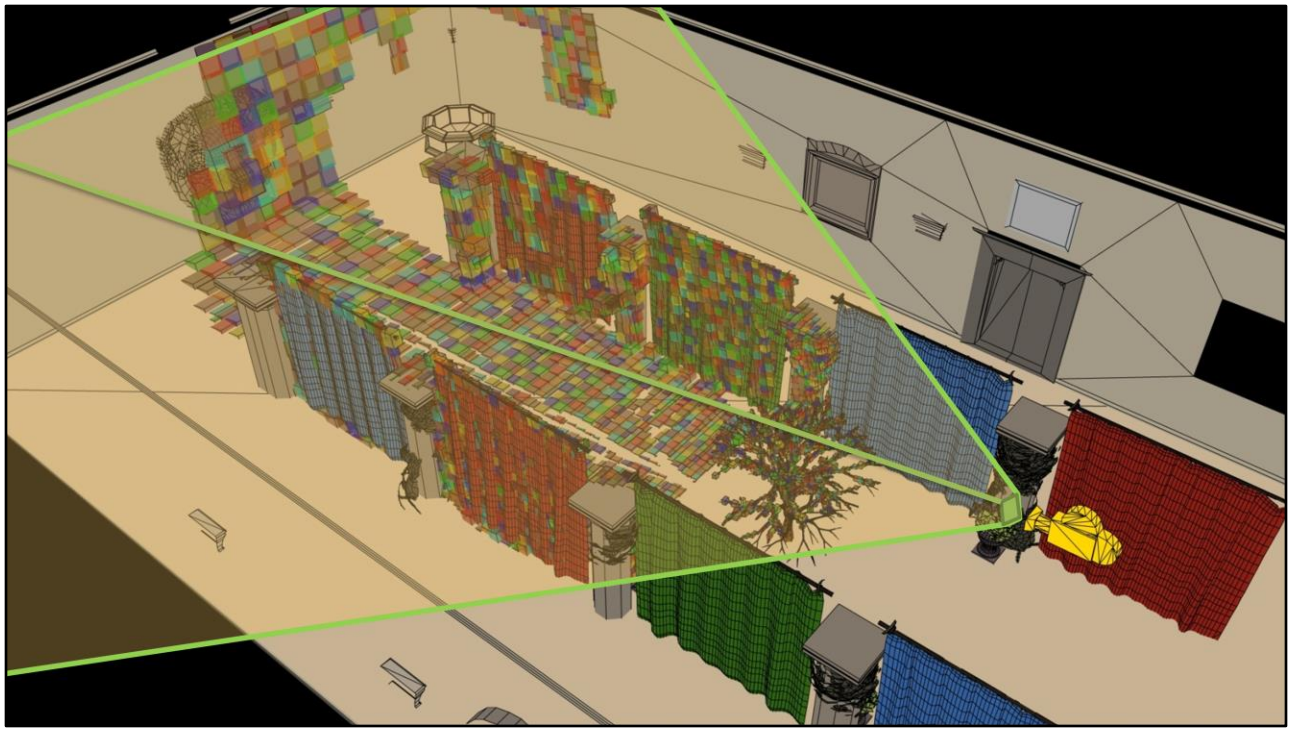
TODO: Remove that square in the middle.



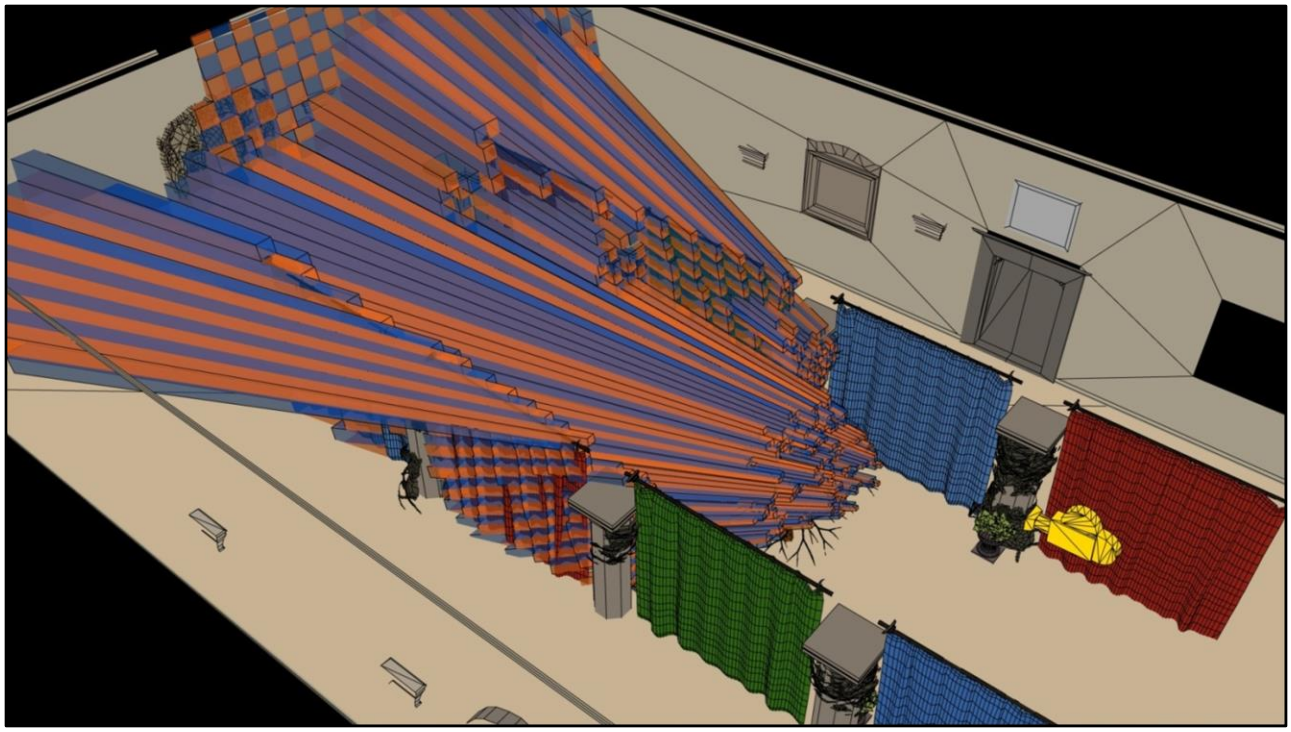
- These are the view space AABBs of the clusters,
- Each cluster represents up to 32x32 in this example, but of course this is tunable.



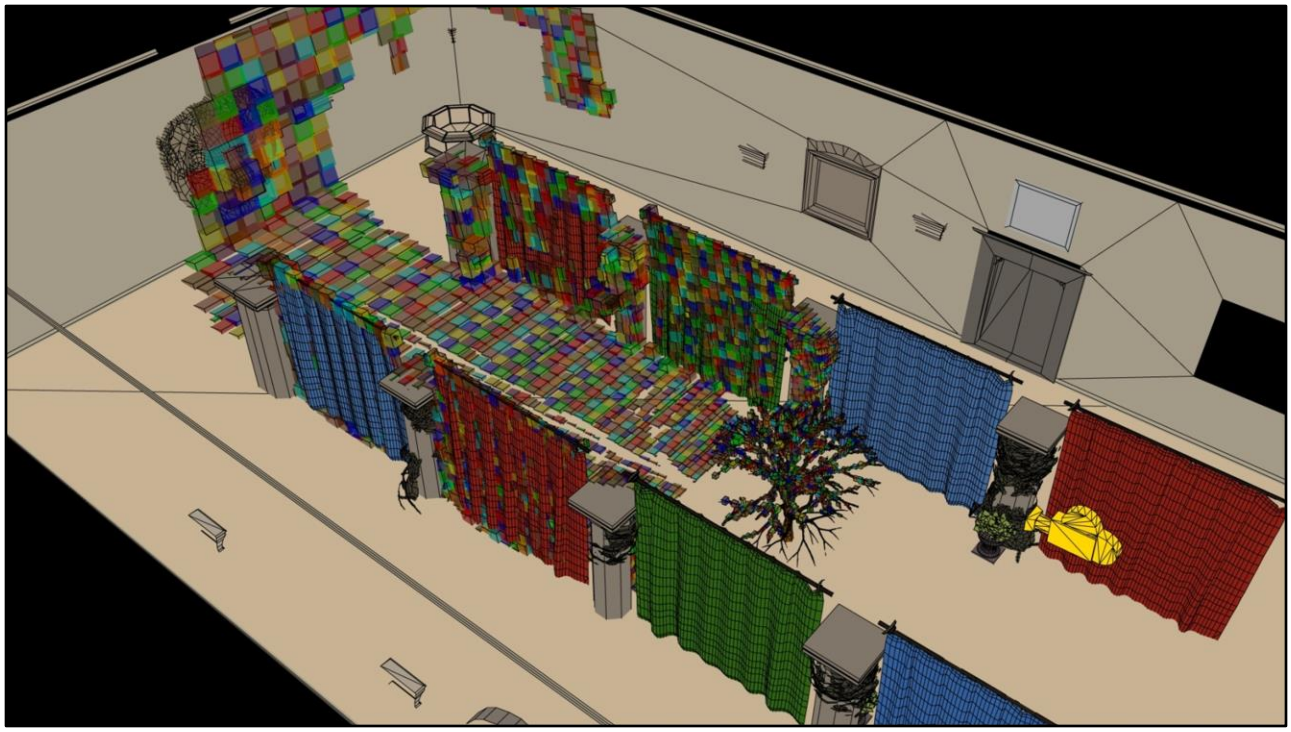
- By changing view, we see that the clusters… (click)



- …approximate the visible scene geometry quite well.
- So we use them instead of directly using the pixels.
- Bringing a couple of orders of magnitude reduction in complexity.



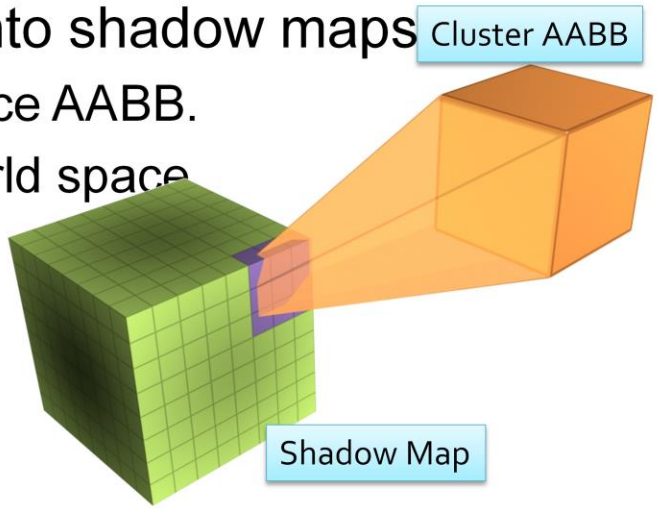
- And, just a side note on why we did not build this algorithm on tiled shading.
- Even with explicit depth bounds, the discontinuities mean that the tiles do not represent the samples very well!



- The difference is quite obvious…

Determining Used Pages

- Project Clusters onto shadow maps
  - Cluster's world space AABB.
  - Shadow map in world space
  - Simple calculation.



- Now however, we must project bounding boxes onto the cube maps instead of points (AKA shadow lookups)
- But by transforming the cluster AABB to world space
- And also keeping cube shadow maps in world space.
- Calculating the projection becomes very simple.

Cube Face +X

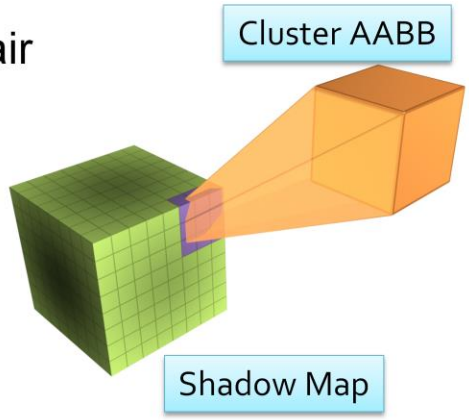
```
float rdMin = 1.0f / max(Epsilon, aabb.min.x);
float rdMax = 1.0f / max(Epsilon, aabb.max.x);

float sMin = min(-aabb.max.z * rdMin, -aabb.max.z * rdMax);
float sMax = max(-aabb.min.z * rdMin, -aabb.min.z * rdMax);

float tMin = min(-aabb.max.y * rdMin, -aabb.max.y * rdMax);
float tMax = max(-aabb.min.y * rdMin, -aabb.min.y * rdMax);
```

- Bounding rectangle on cube face.
- Repeat 6 times, slightly different.

- This code is what we use to compute the projection on one cube face.
- And as you can see, it is not extremely complex.
- We repeat this, slightly differently for each face and get a rectangle in texture coordinates for each.
- This is then converted to a bit mask, with a single bit for each tile or page in the virtual cube map.

Process

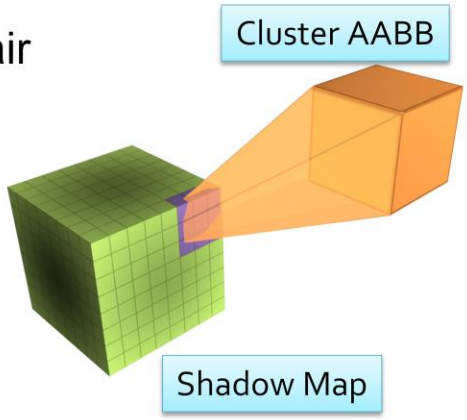
- In parallel, for each cluster/light pair
  - Project cluster AABB onto light SM.
  - Update bit masks for each light.
    - One bit/physical page.
    - 32x32x6 bits = 768 byte / light.
  - Atomic reduction atomicOr
  - <0.25ms / 180k cluster/light pairs



- We compute this, by running a cuda thread for each cluster/light pair.
- In the kernel we calculate the projection just shown
- As the mask is stored for each light and thus referenced by many cluster light pairs,
- It is updated using atomicOr.
- This is a pretty quick pass, <0.25ms, 180k light/cluster pairs.

Process

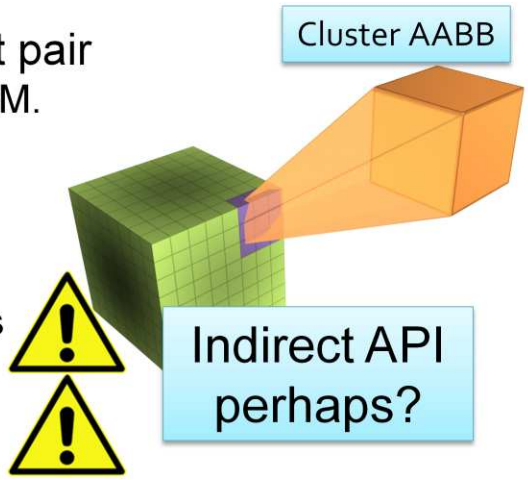
- In parallel, for each cluster/light pair
  - Project cluster AABB onto light SM.
  - Update bit masks for each light.
    - One bit/physical page.
    - 32x32x6 bits = 768 byte / light.
  - Atomic reduction atomicOr
  - <0.25ms / 180k cluster/light pairs
- Copy masks back to host.
- Commit physical pages (host)
  - glTexPageCommitmentARB



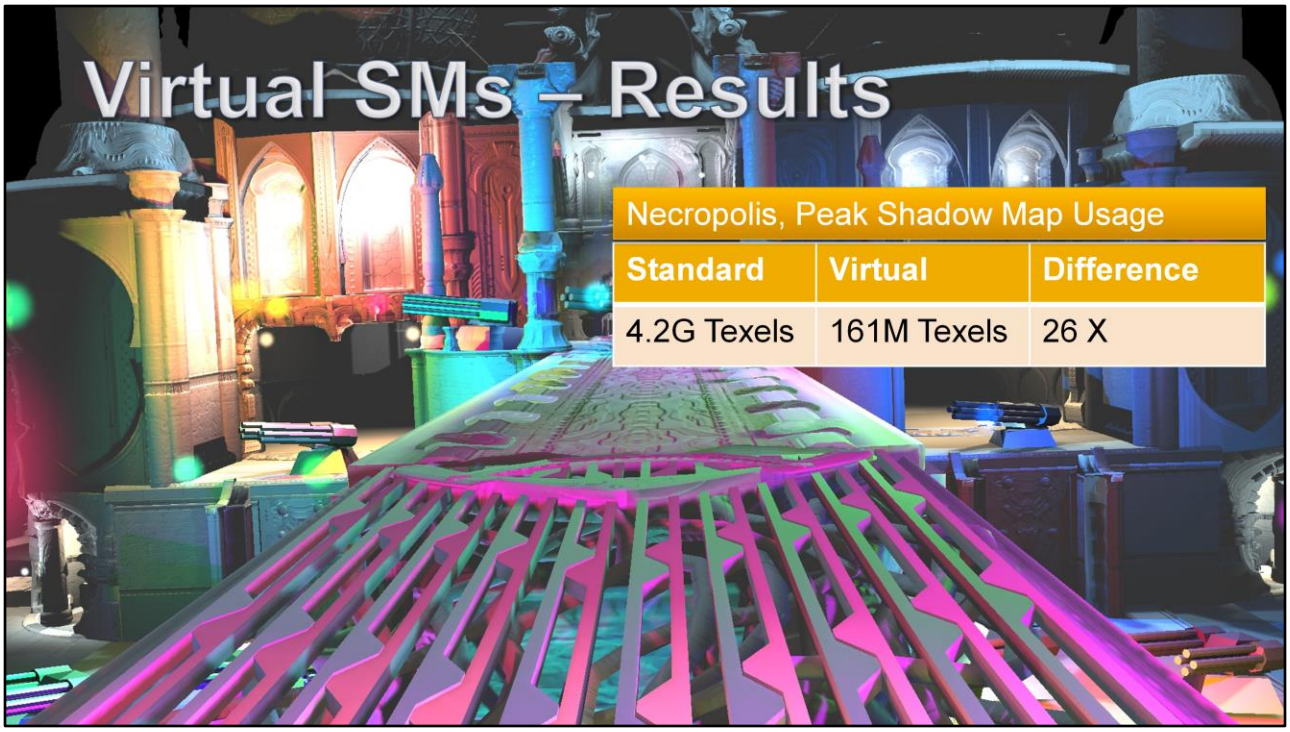
- The masks are copied back to the host, and used to commit pages for the shadow map textures.

Process

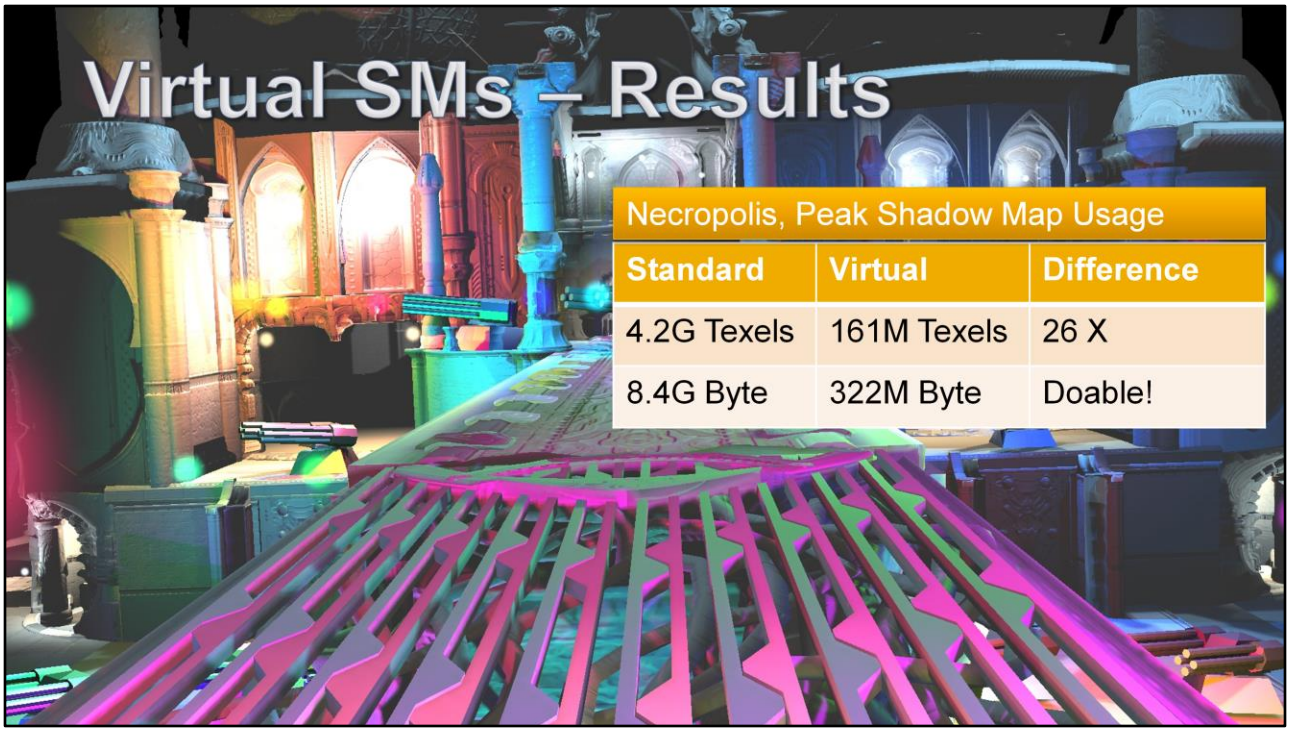
- In parallel, for each cluster/light pair
  - Project cluster AABB onto light SM.
  - Update bit masks for each light.
    - One bit/physical page.
    - 32x32x6 bits = 768 byte / light.
  - Atomic reduction atomicOr
  - <0.25ms / 180k cluster/light pairs
- Copy masks back to host.
- Commit physical pages (host)
  - glTexPageCommitmentARB



- Warning bells
- This is potential stall waiting to happen, and also forces an API call per page commit.
- We'll come back to this problem later.



- So we might ask, is it worth doing virtual shadow maps?
- Well, for the necropolis scene, there is a factor 26 improvement,
- and



- In other terms, this means the difference between impossible on a current console
- And something we might consider.

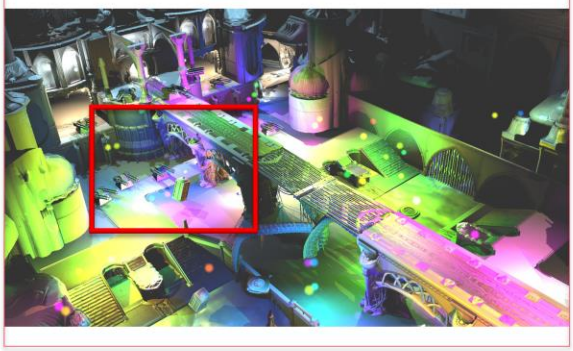
Quality vs. Memory Usage

- Quite uniform quality.
- Simple to reduce
  - Undersampling parameter.
- Control memory use.
- Could be done dynamically.
  - Guarantee peak memory use?

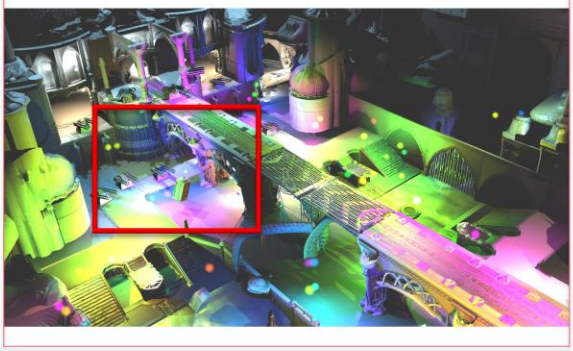
- Our method achieves quite uniform shadow quality,
- This means we can control quality and thus memory usage with a global parameter.
- This allows more flexibility in memory use while maintaining uniform quality.
- An interesting idea is to do this dynamically, to ensure a certain memory budget.
- Should be possible as it is very quick to work out memory usage from the used pages and resolutions.

Quality vs. Memory Usage

4X UNDERSAMPLING ~80MB

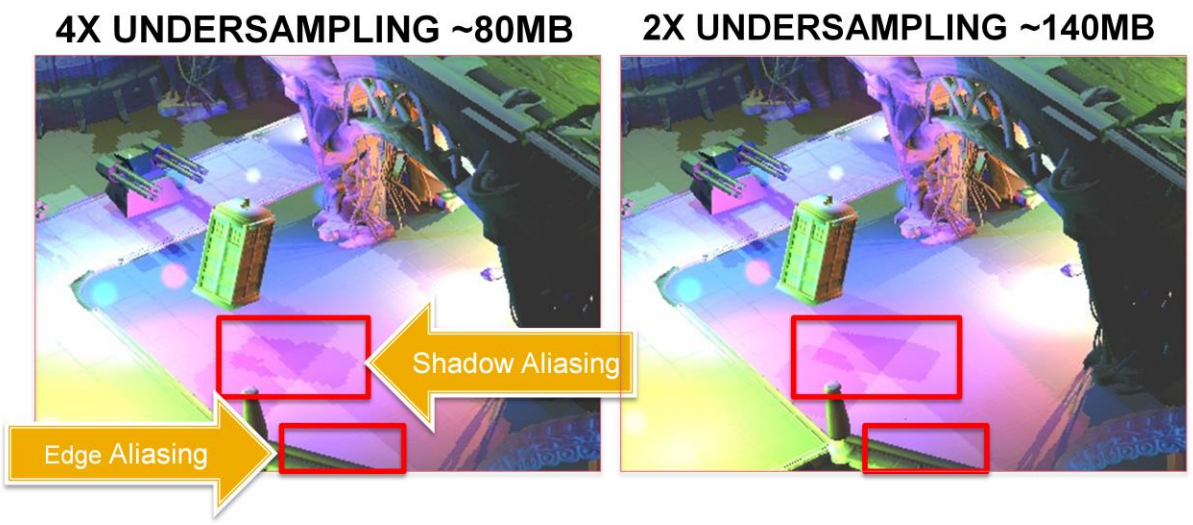


2X UNDERSAMPLING ~140MB



- Here is a shot showing this, with 2x or 4x global reduction in shadow map resolution.
- Shown are the corresponding peaks in shadow map memory usage.
- Recall that the peak is 322MB without reducing quality.

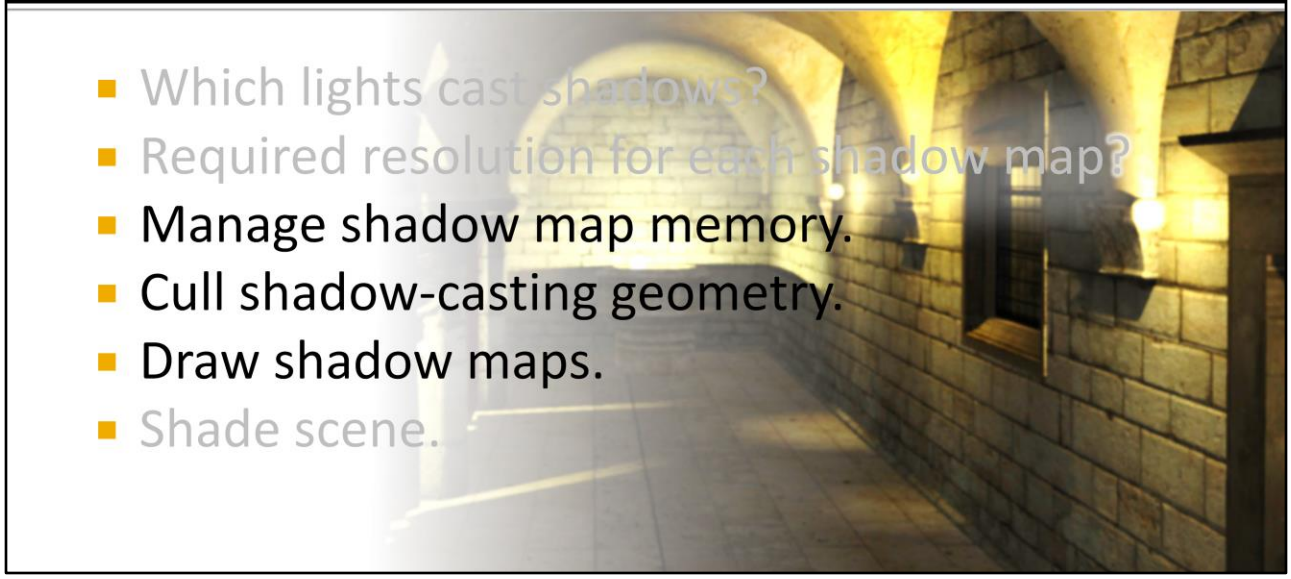
Quality vs. Memory Usage



- Zooming in, we see that the shadow aliasing is about the expected level when compared to edge aliasing in the image.
- This indicates that our simple scheme for calculating the required resolution yields reasonable results.
- With filtering, this can be acceptable, especially as it allows *uniform* quality rather than unevenly allocating shadow maps.
- Memory usage is, as you see rather lower.

Problem breakdown

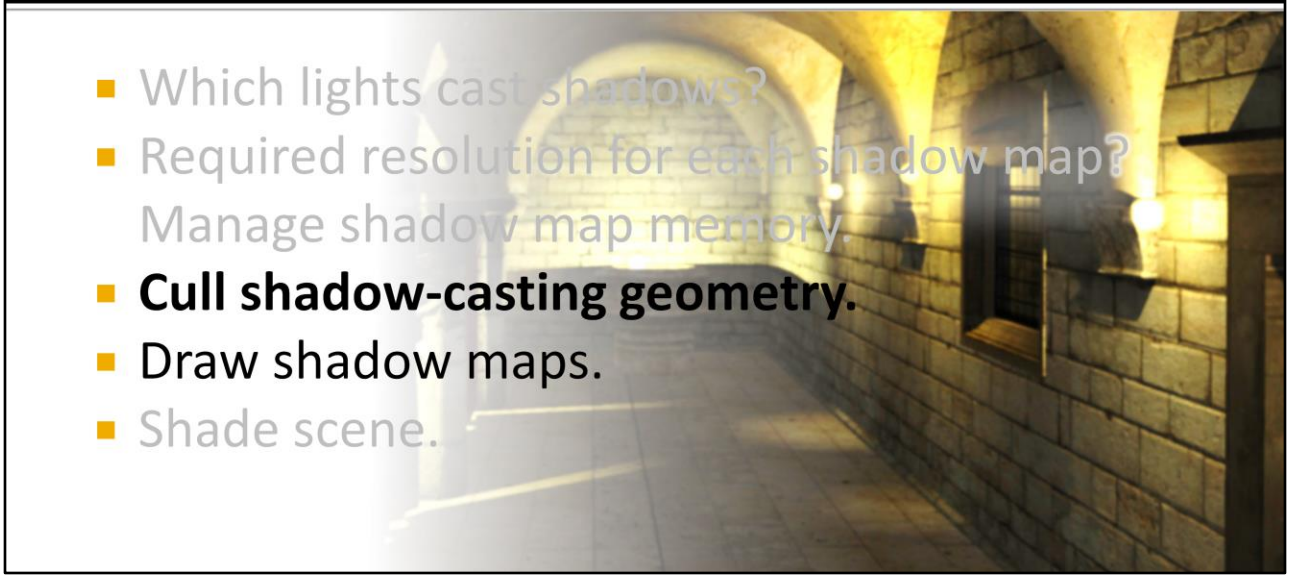
CHALMERS



- So that takes care of memory management, and we can now allocate shadow maps to draw into.

Problem breakdown

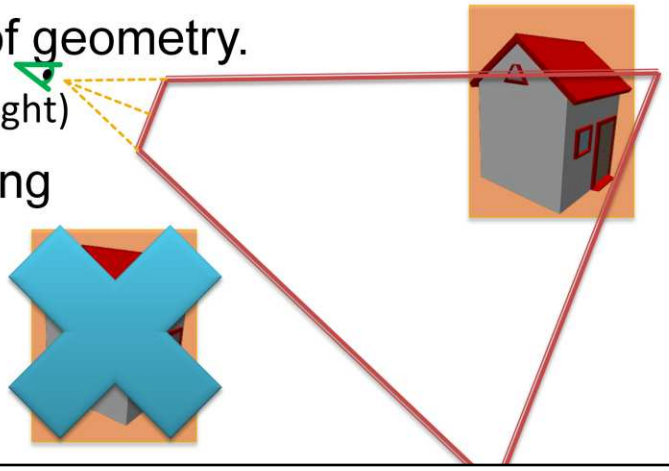
CHALMERS



- Next must rasterize triangles into the shadow maps.
- To do this efficiently requires culling, as with any rasterization only more so.

Basic Culling

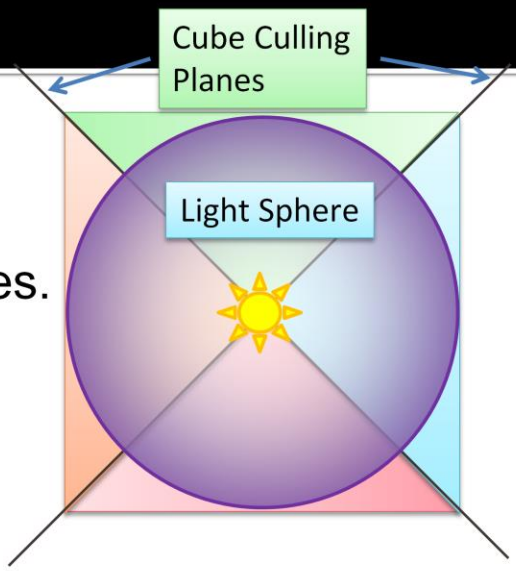
- Just like View Frustum Culling
  - Cull chunks of geometry.
    - Eye/Camera (Light)
  - Using bounding boxes.



- This process is just like normal view frustum culling
- In that we are trying to get rid of geometry that is not visible,
- And that we do this by testing bounding volumes of chunks or batches of triangles.

Basic Culling

- *Not* like VFC
  - Hundreds of lights.
  - Mostly short view volumes.
    - Limited light range.
  - Omni-directional lights.
    - 6 ***adjacent*** frustra.
    - Sharing planes.



- What is not like view frustum culling is that we need to perform hundreds of these tests.
- The view volumes are quite small, or short, given the limited range of the lights
- There are 6 adjacent frustra sharing planes.
- This adjacency means we can share calculations.

```
int getCubeFaceMask(float3 cubeMapPos, Aabb aabb)
{
    float3 planeNormals[] = { {-1,1,0}, {1,1,0}, {1,0,1}, {1,0,-1}, {0,1,1}, {0,-1,1}};
    float3 absPlaneNormals[] = { {1,1,0}, {1,1,0},{1,0,1}, {1, 0,1}, {0,1,1}, {0,1,1}};

    float3 center = aabb.getCentre() - cu    os;
    float3 extents = aabb.getHalfSize();

    bool rp[6];
    bool rn[6];

    for (int i = 0; i < 6; ++i)
    {
        float dist = dot3(center, planeNormals[i]);
        float radius = dot3(extents, absPlaneNormals[i]);
        rp[i] = dist > -radius;
        rn[i] = dist < radius;
    }

    int fpx = rn[0] && rp[1] && rp[2] && rp[3] && aabb.max.x > cubeMapPos.x;
    //…same for other faces…
    return fpx | (fnx << 1) | (fpy << 2) | (fny << 3) | (fpz << 4) | (fnz << 5);
}
```

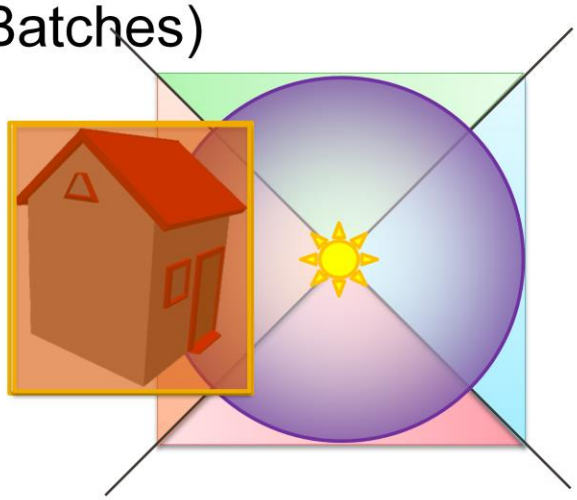
Literal constants, all ones or zeroes.

If unrolled, very few operations left!

- Here is the somewhat condensed code we use to calculates the culling mask, with a bit for each cube face.
- This a very efficient, testing only 6 planes for six frustums. (click)
- Especially as the plane equations are all ones and zeroes, (click)
- Which means that if the loop is unrolled, most of this code just goes away!
- I think this is a rather big advantage with cube maps, over using separate frustums (As done in [6]).
- This efficiency is especially important given that we will be culling a lot more objects than normal culling!
- How so, I hear you say?
- Glad you asked!

Basic Culling

- Fine Granularity (Batches)
  - Must be small.
  - Match size of light.

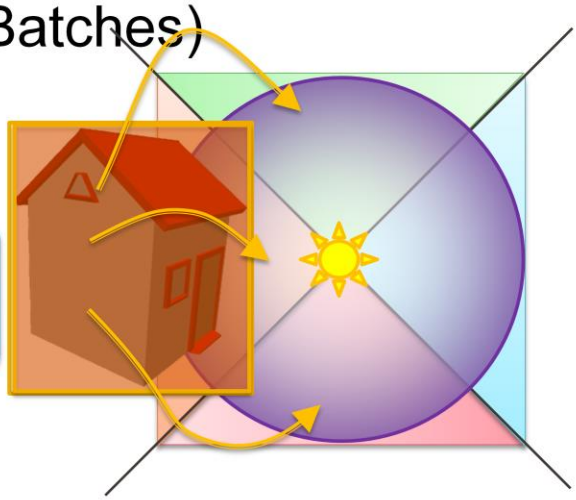


- You see, to enable efficient culling, batches must be small,
- Intuitively, for any culling, the optimal size of batches correlates to the size of the frustums.

Basic Culling

- Fine Granularity (Batches)
  - Must be small.
  - Match size of light.

1000 Tris x 3 Cube faces
= 3k

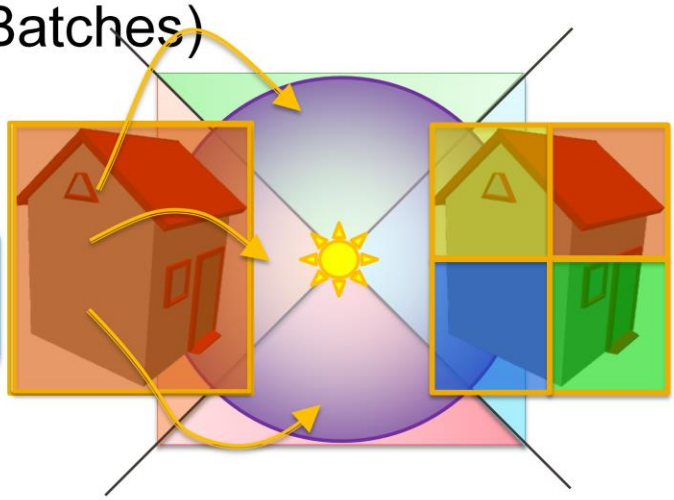


- If batches are too large, the triangles get replicated into most cube faces replicated.

Basic Culling

- Fine Granularity (Batches)
  - Must be small.
  - Match size of light.

1000 Tris x 3 Cube faces
= 3k



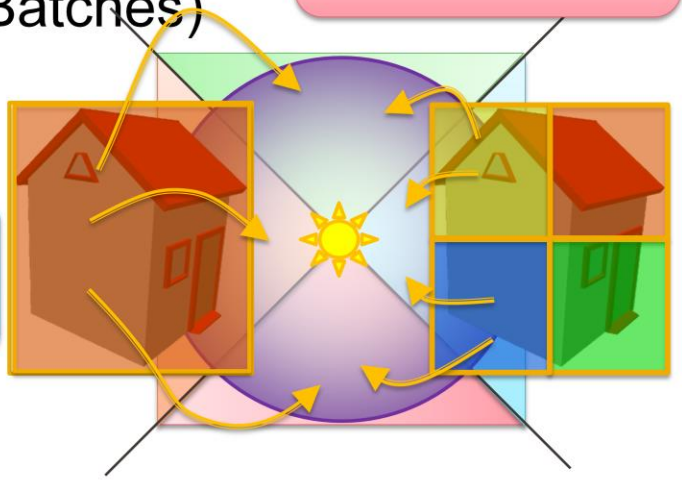
- Smaller batches, enables

Basic Culling

- Fine Granularity (Batches)
  - Must be small.
  - Match size of light.

1000 Tris x 3 Cube faces = 3k

(250 Tris x 2) x 2 = 1k



- More precise culling, and thus fewer triangles drawn.
- So we are trading increased culling work for fewer triangles drawn.
- Triangle drawing is the biggest performance bottleneck so this is important to be able to tune.

Batches

- Batch = up to 128 triangles
  - AABB.
  - Index to transform, triangles.
- Constructed offline
  - Group Coherent Triangles.
    - Same transform(s).
  - Flat array.

```
struct Batch
{
  Aabb aabb;
  int tris;
  int offset;
  int transformInd;
};
```

- We used batches of up to 128 triangles,
- in practice they average around 68 triangles.
- A batch is represented by an AABB and a list of triangles.
- The batches are constructed in a pre-process, that builds a tree using agglomerative clustering (see the paper section 6.2.1).
- Note that the quality of the batches is fairly important for good performance.
- The batches are stored in a flat array that is loaded into the runtime.

Culling Efficiently

- Q: Is your acceleration structure bad enough?

- For efficient culling, we obviously need a hierarchy.
- The perhaps somewhat controversial question posed here at first appears the reverse of what we would normally ask ourselves…

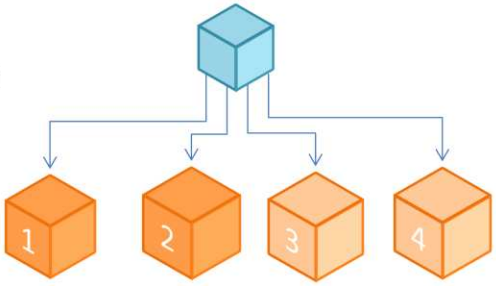
Culling Efficiently

- Q: Is your acceleration structure bad enough?
  - Need to spend just enough time building it.
  - Good and expensive?
  - Fast and Crappy?
  - Trade-off studied in ray tracing [3]
    - Karras et Aila, 2013, 'Fast Parallel Construction of High-Quality Bounding Volume Hierarchies'

- The important thing is to balance the time spent building and traversing an acceleration structure.
- This trade-off has been studied by Karras and Aila in context of ray tracing, and is a very interesting read.
- In short, it suggests to me that the acceleration structure for just a few thousand box queries must be pretty bad to be worth building.

Batch BVH

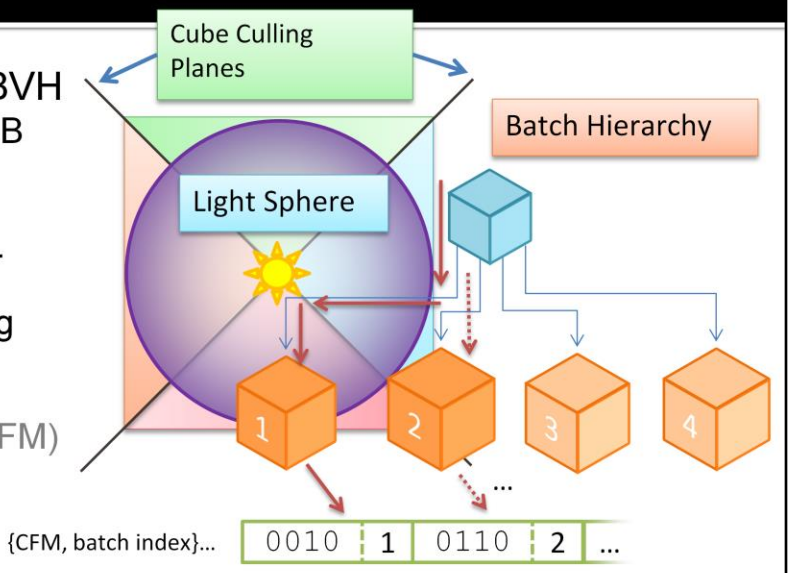
- Constructed online
  - Each frame.
  - Batch AABBs updated from vertices.
  - Simple 32-way full BVH.



- We used a very simple, full, 32-way BVH which is completely rebuilt each frame.
- There are more details on this structure in our papers, and even CUDA code online in the clustered forward demo.
- This is in no way the best possible structure, or even the fastest to build, but it has served us well.

Culling

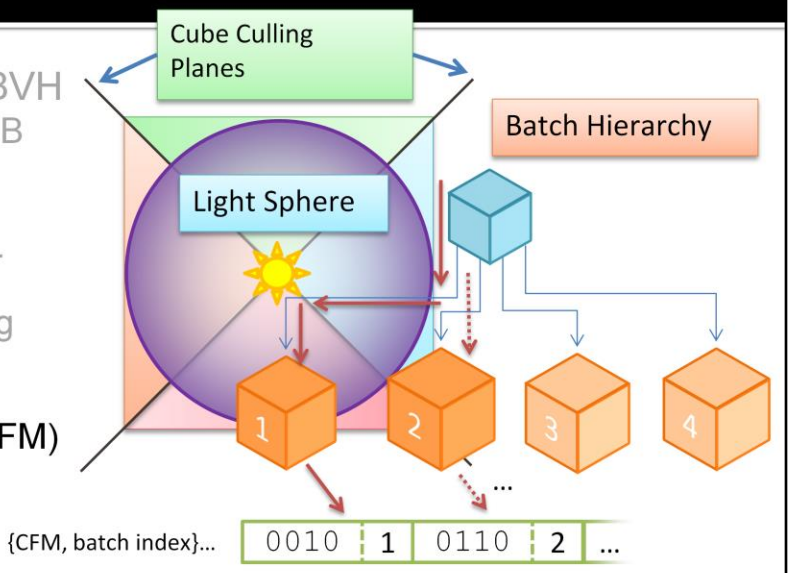
- In parallel, traverse BVH
  - Light sphere vs AABB test
- At leaves (batches)
  - Test Light sphere vs. AABB.
  - Test AABB vs. culling planes.
- Output
  - Cube Face Mask (CFM)
  - Batch index



- Here we parallelize of the lights, and each light traverses the hierarchy to find the batches that overlap the sphere
- These batches are those that may produce a shadow if drawn into the shadow map.

Culling

- In parallel, traverse BVH
  - Light sphere vs AABB test
- At leaves (batches)
  - Test Light sphere vs. AABB.
  - Test AABB vs. culling planes.
- Output
  - Cube Face Mask (CFM)
  - Batch index

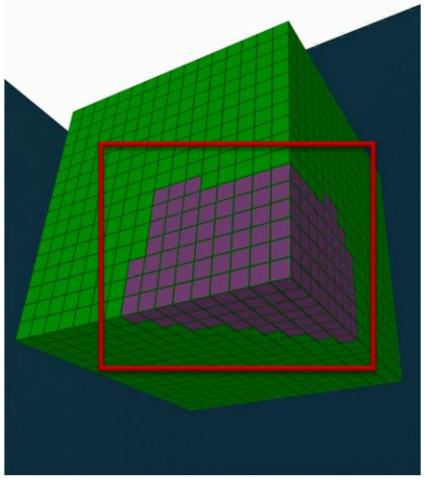


- This produces a list for each light of pairs of cube face masks and batch indexes
- The Cube Face Mask is a bit mask where each bit indicates if it overlaps a certain cube face.
- This tells us what batches to draw to which cube faces of each light.

Better Culling

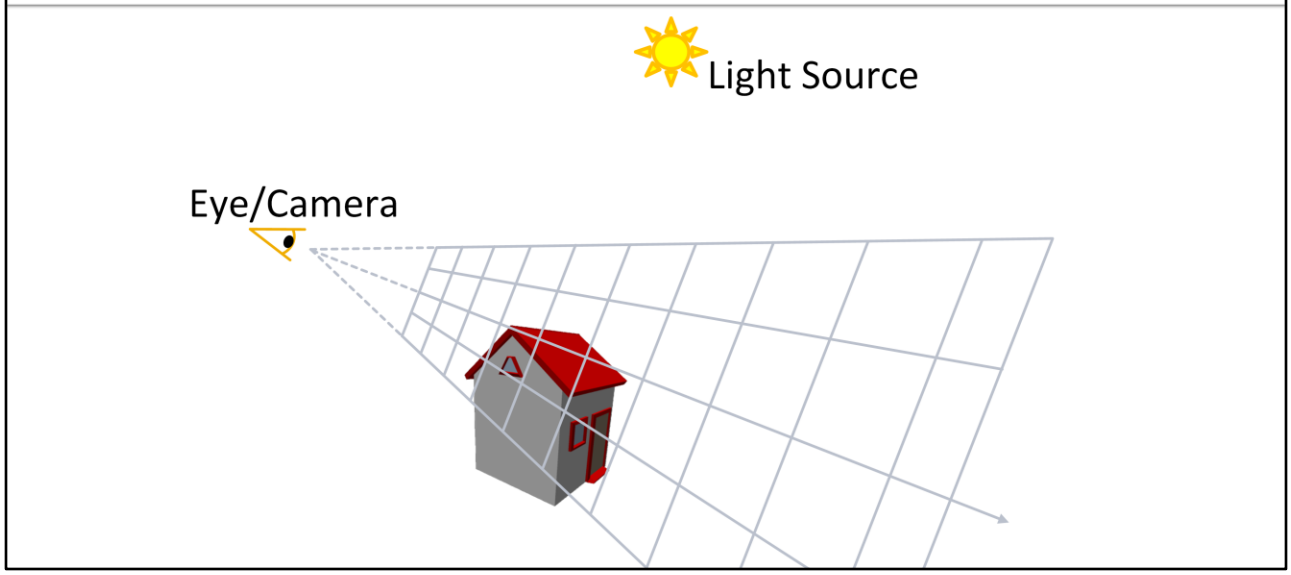
0010	1	0110	2	…

- Shadow receivers
  - I.e. view samples
    - I.e. Clusters
- Why draw where no samples will be taken?
- Example:
  - Purple = committed.
  - Why draw outside?



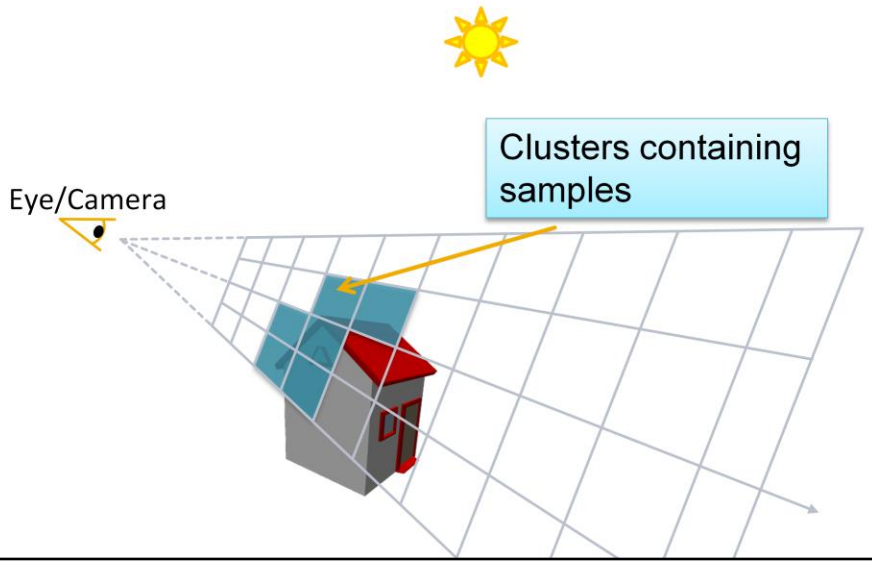
- Now, we also wish to take advantage of the sparsity of the shadow maps
- …to avoid drawing geometry to parts of the shadow map which will not be sampled, or indeed stored!

Better Culling



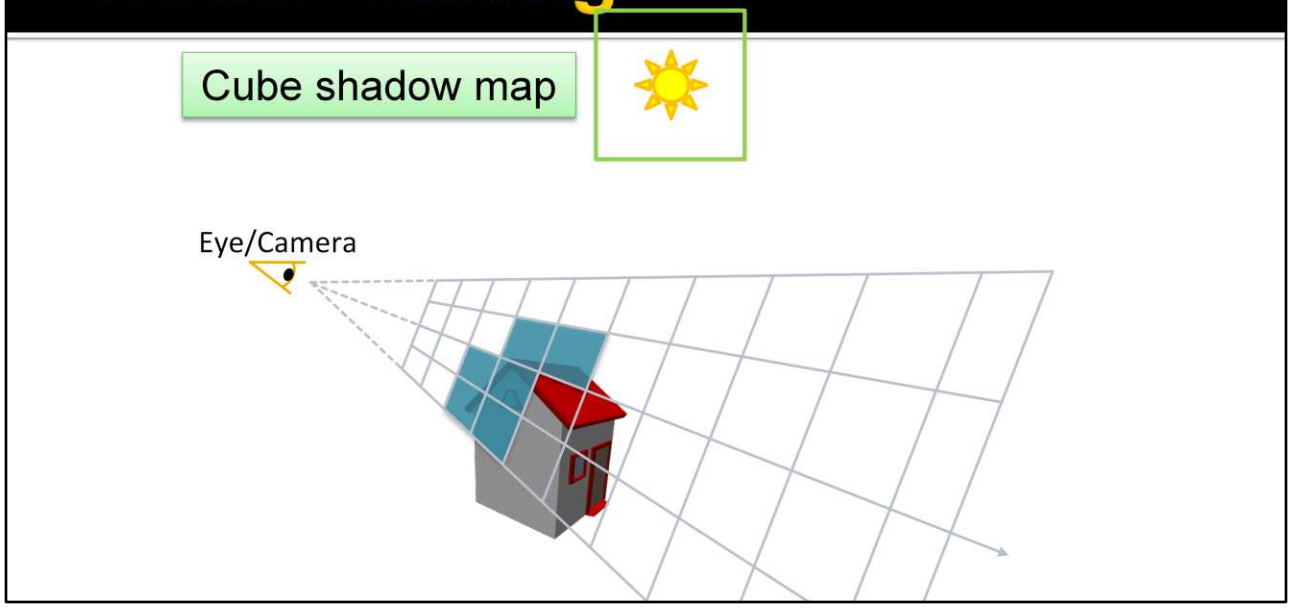
- In this example we have a camera, looking onto the little house, and an overhead light source

Better Culling



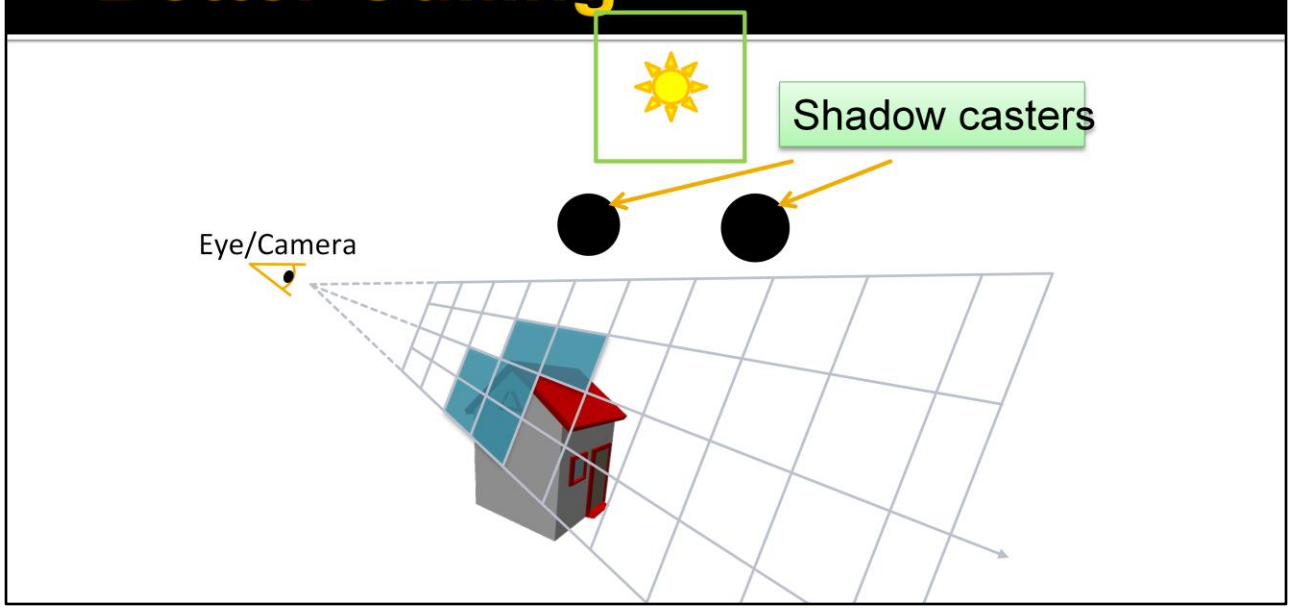
- Clusters that contain visible samples are shown here,
- They represent the sample points that shadow may be queried for, or the shadow recievers

Better Culling



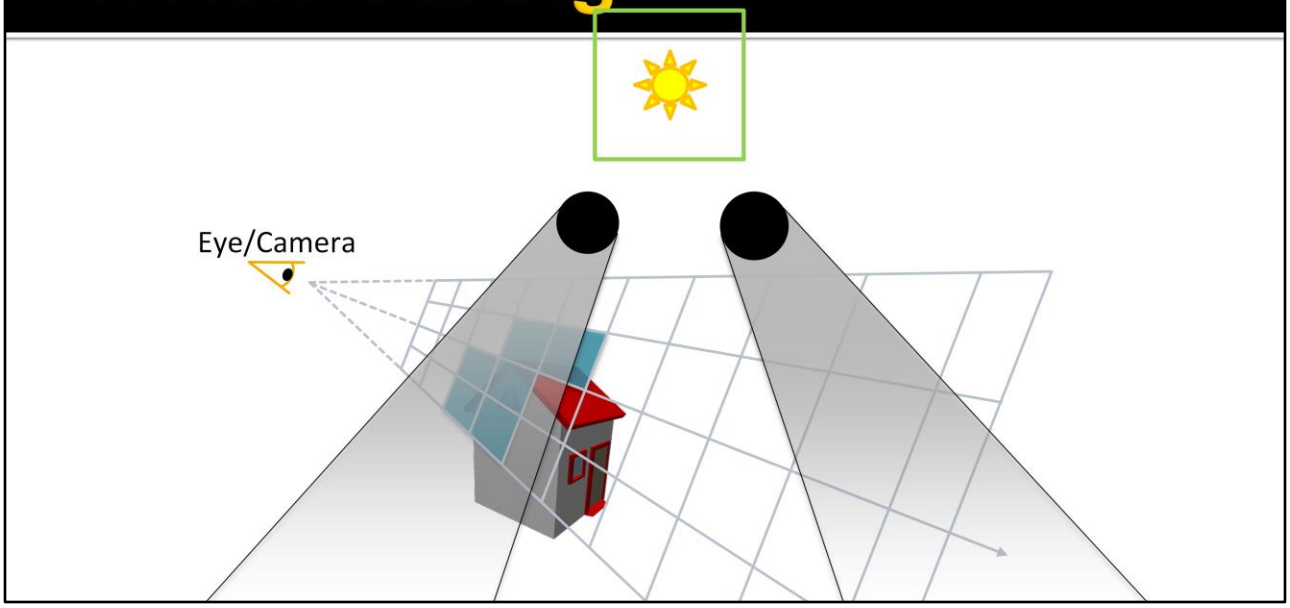
- This box represents the cube shadow map for the light.

Better Culling



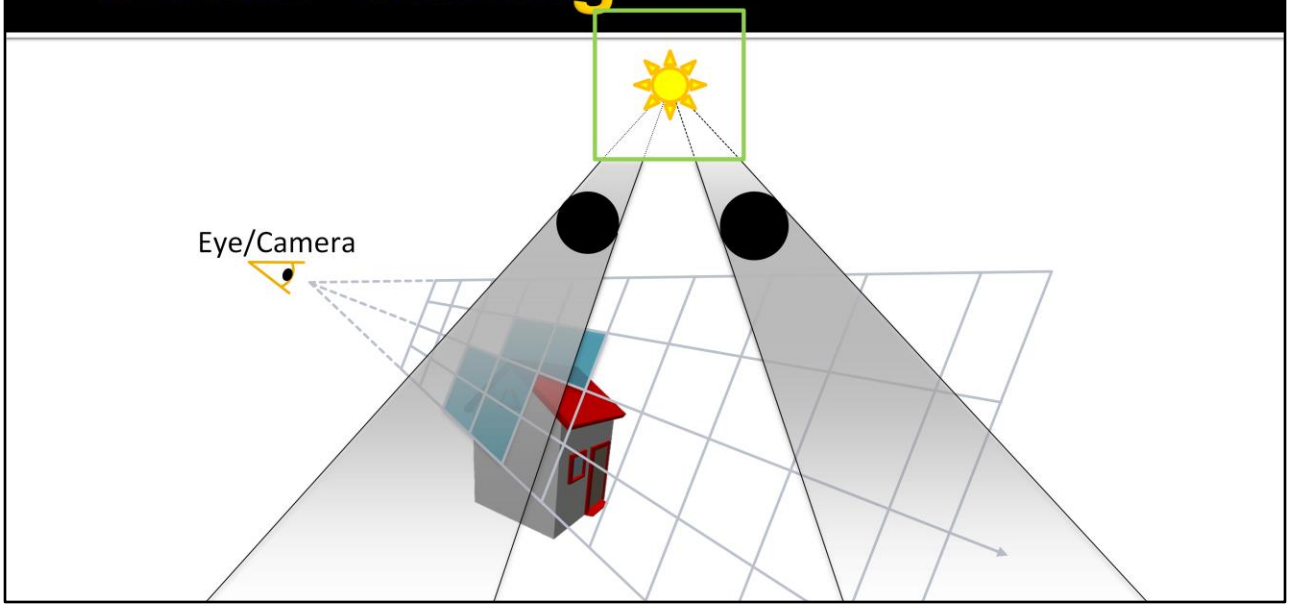
- Here are two shadow casters.
- Note that they are outside of the view volume, and so have no clusters associated…

Better Culling



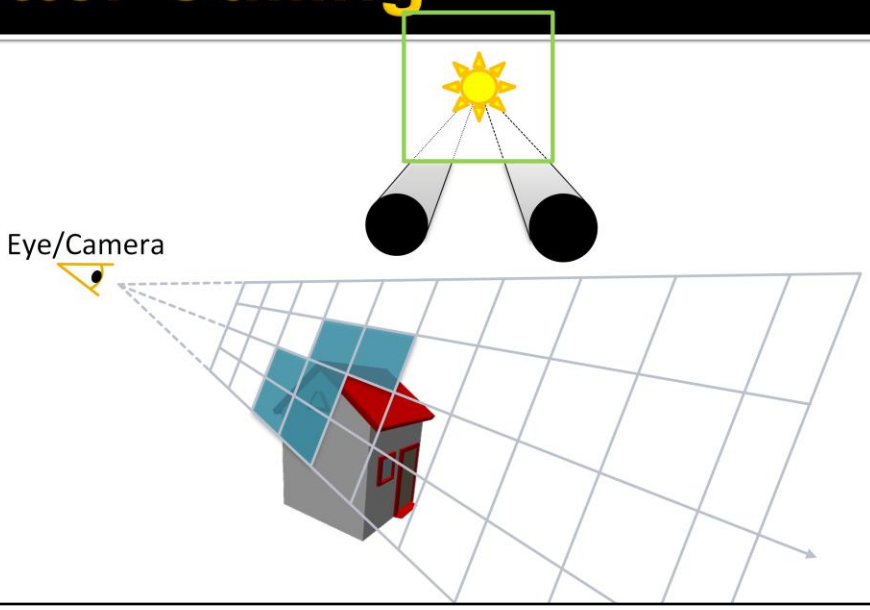
- But cast their shadow through the view volume.
- So we're interested in finding out what shadows affect the visible samples,
- and thus determine if the shadow caster need to be drawn.

Better Culling



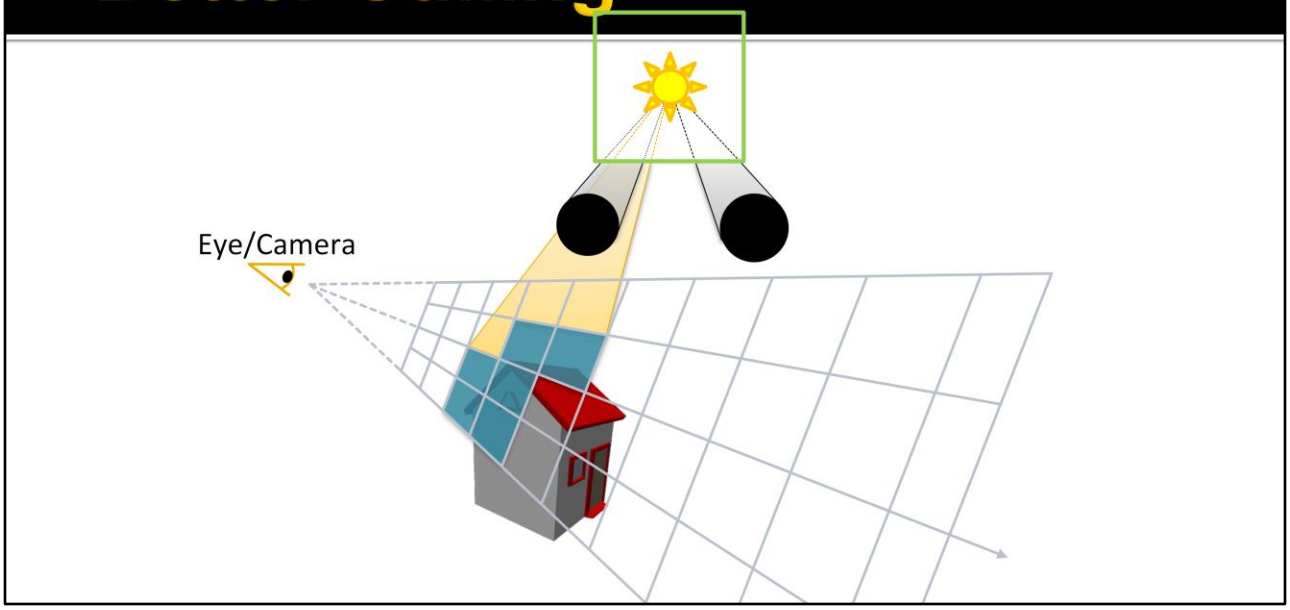
- Instead of doing some crazy thing, like shadow volumes.
- We can figure this out by projecting the shadow caster onto the light source

Better Culling



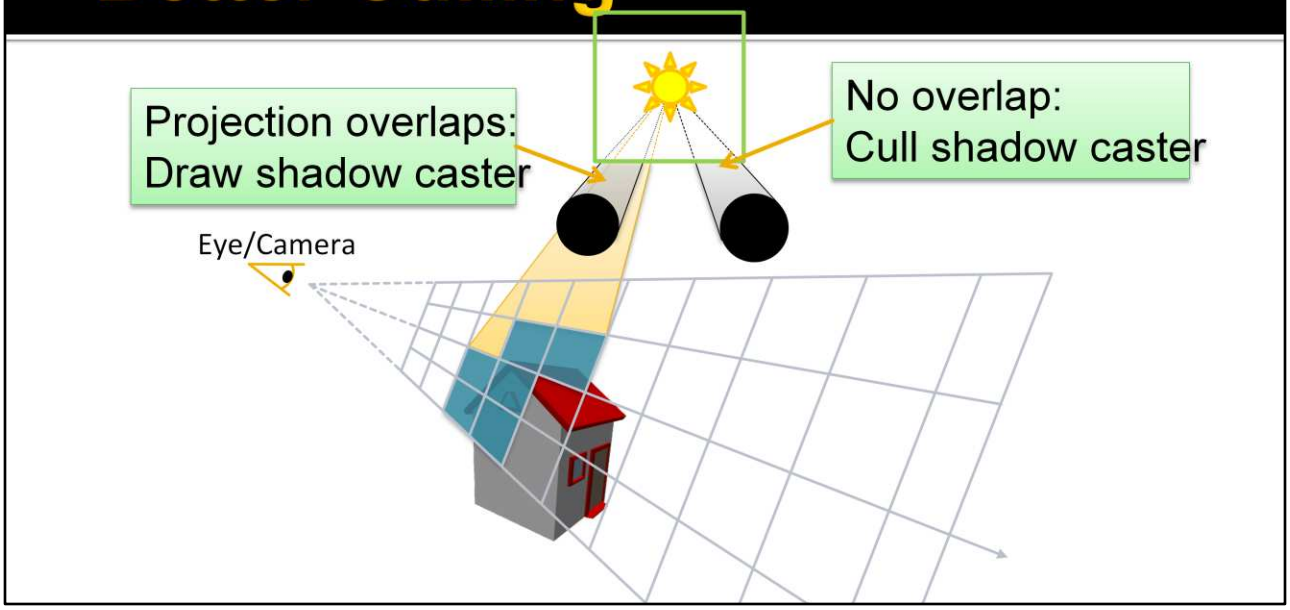
- Like so.
- This gives us these two intervals on the cube face.

Better Culling



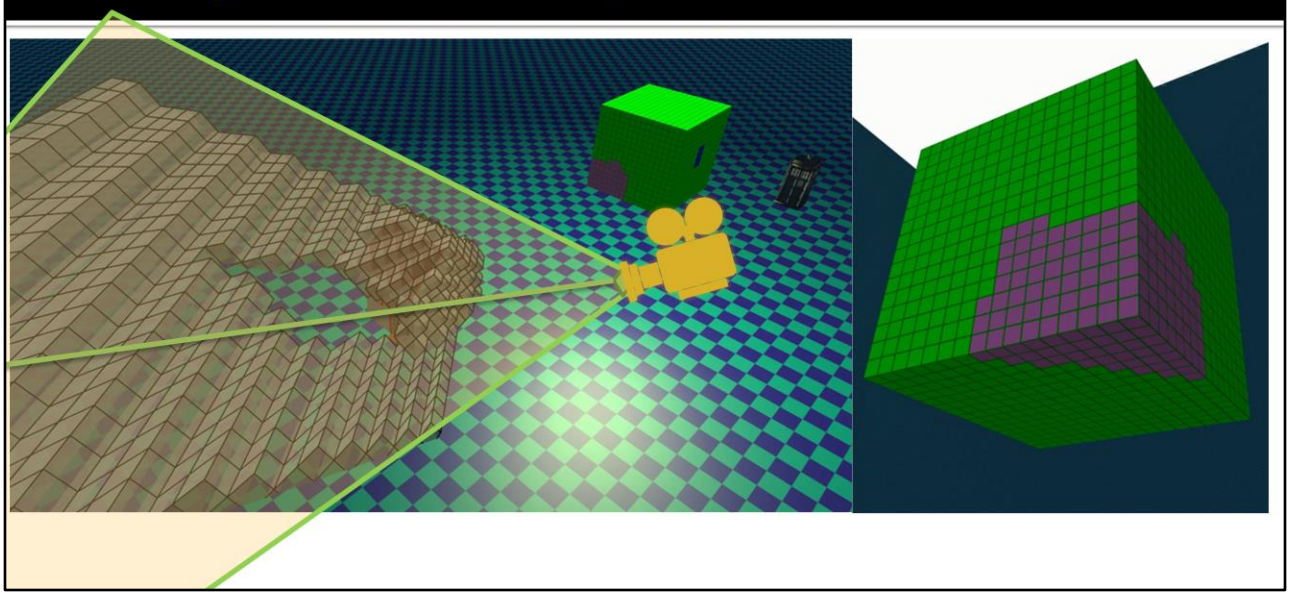
- Doing the same thing for the clusters
- Gives another interval here

Better Culling



- If the projection of a shadow caster overlaps that of some clusters, i.e. shadow reciever,
- …it needs to be rendered into this shadow map
- Else, it doesn't.
- Note that the clusters don't need to form a range for this to work, any overlap will do.

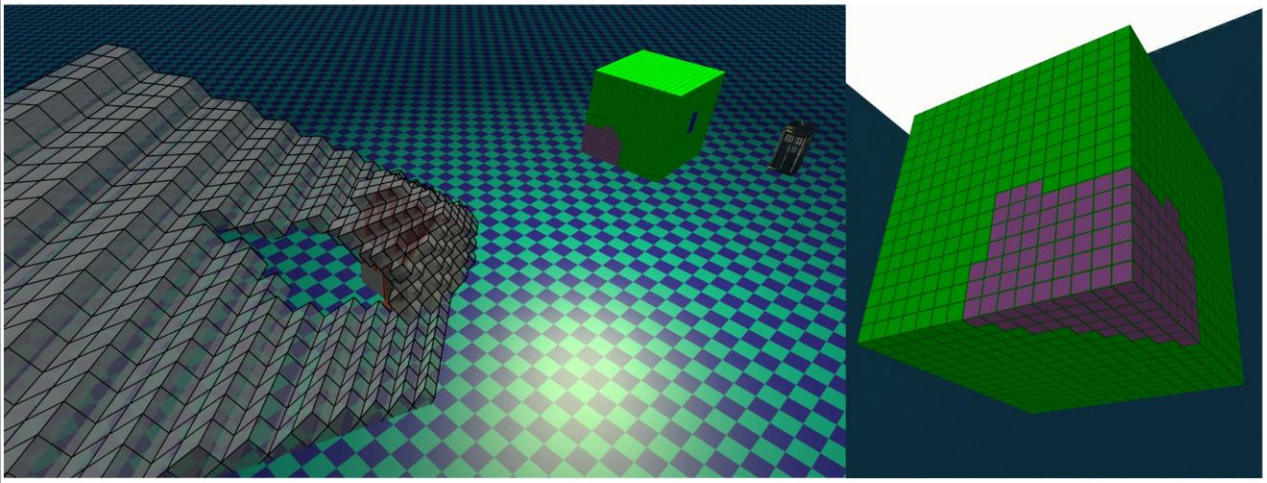
Projection Maps



- we implement this idea by re-using the quick projection and bit-rasterization used for virtual page allocation
- The yellow camera and frustum here illustrates the view that produced the clusters on the building and ground.
- The actual rendered, overhead, view is to illustrate the process.
- We first project the clusters onto the cube map, producing a bit mask, the bits set in this process are shown in purple.
- Next, for each geometry batch in turn, we perform the same projection,
- Here we only have one batch: the orbiting tardis, shown in blue. Note that the batch is outside the view frustum all the time,
- but that it can cast shadow from the light into the scene.
- Then we compare the masks and if there is any overlap, shown in yellow,
- then the geometry may cast a visible shadow and must be drawn.

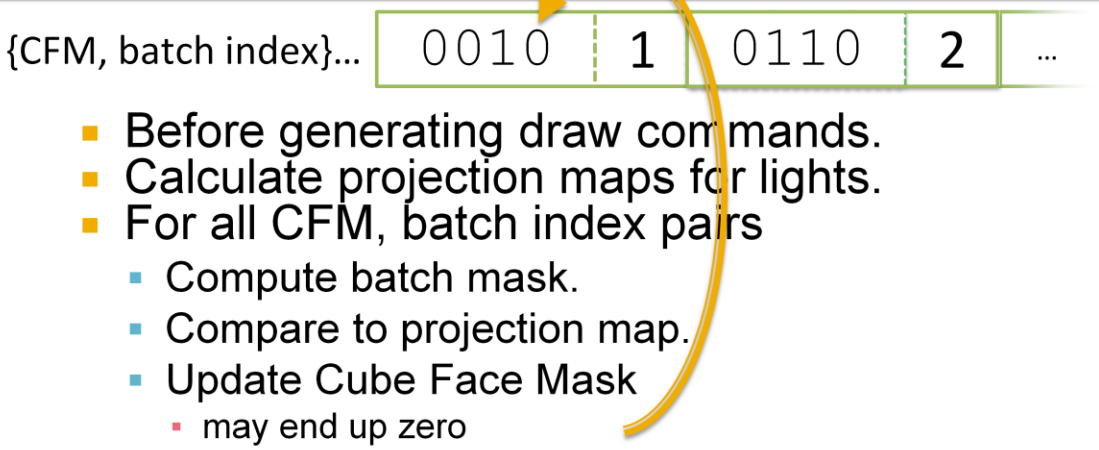
Projection Maps

0010	1	0110	2	…



- So as we see, most of the time the tardis can be culled, despite always being in the field of view of at least one cube face.

Prune Batches

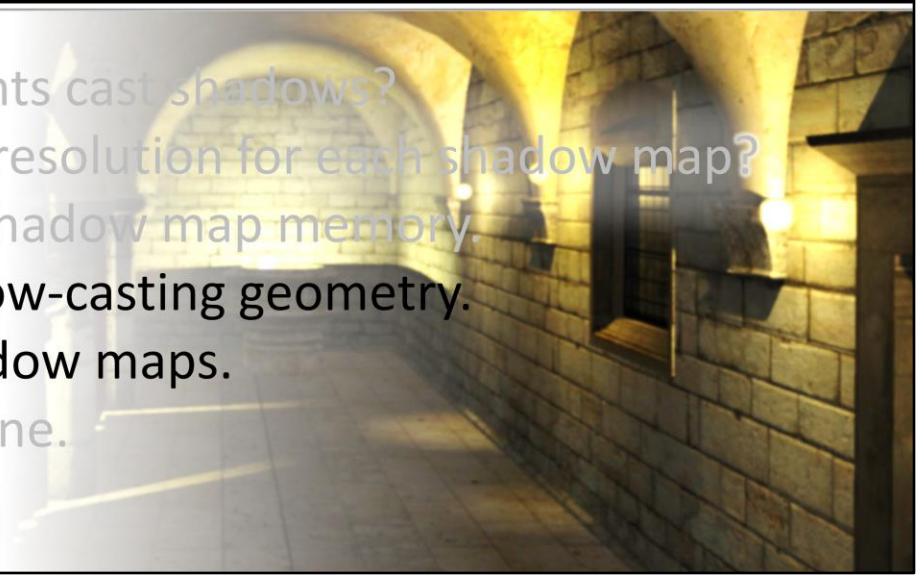


- We implement this with a pass over all the results from the previous step.
- The kernel just loads the light and, cluster and computes the projection map for the cluster
- Then the projection map of the light (i.e., the projection of the clusters) is checked against the cluster, cube face by cube face.
- For any where there is no overlap, the cube face mask bit is cleared, meaning it will not be drawn.
- Finally the updated CFMs are stored back, now usually with fewer bits set, and sometimes zero.

Problem breakdown

CHALMERS

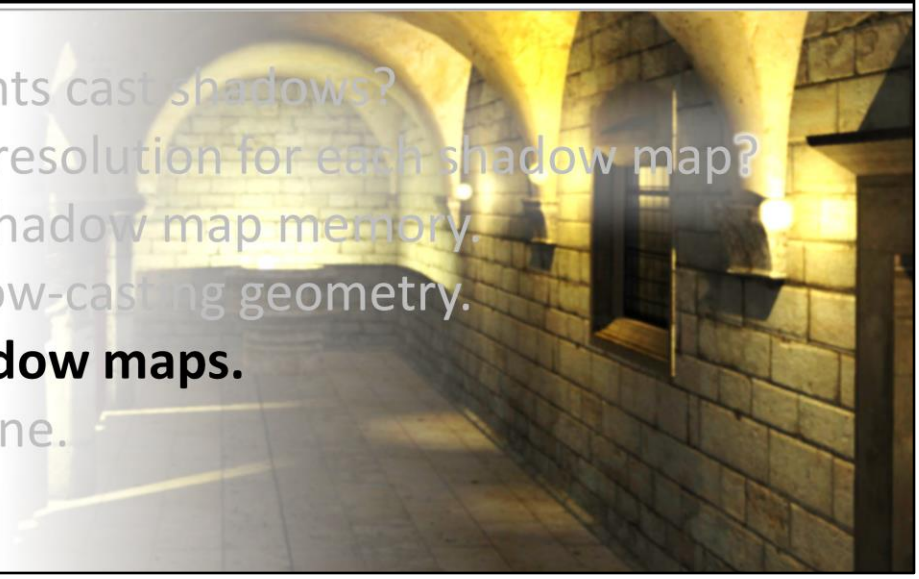
- Which lights cast shadows?
- Required resolution for each shadow map?
- Manage shadow map memory.
- Cull shadow-casting geometry.
- Draw shadow maps.
- Shade scene.



Problem breakdown

CHALMERS

- Which lights cast shadows?
- Required resolution for each shadow map?
- Manage shadow map memory.
- Cull shadow-casting geometry.
- **Draw shadow maps.**
- Shade scene.



Drawing

- As all the culling is performed on the GPU, using CUDA, make use of modern OpenGL and build draw commands also on the GPU
- Then we just call multi draw indirect once for each cube map (as they are separate render targets)

Drawing

{CFM, batch index}…


```
struct DrawElementsIndirectCommand
{
  uint32_t count;
  uint32_t instanceCount;
  uint32_t firstIndex;
  uint32_t baseVertex;
  uint32_t baseInstance;
};
```

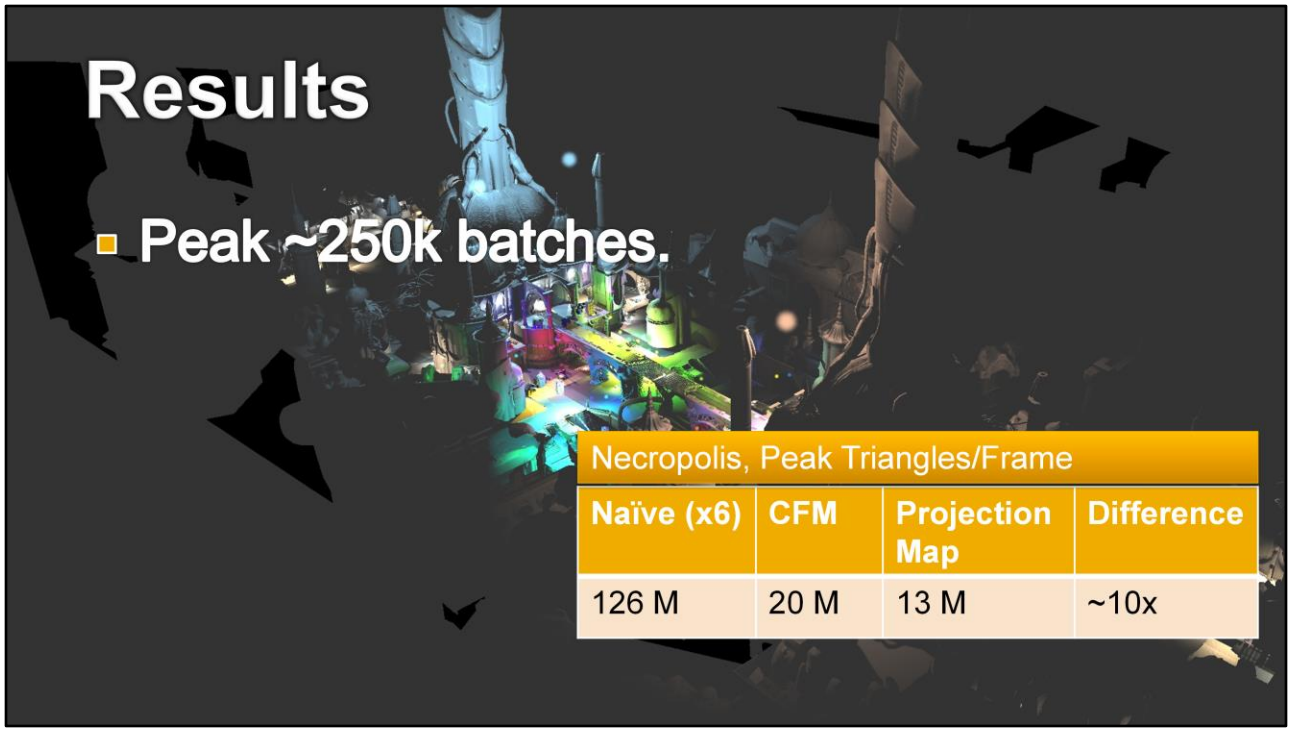
- The draw commands look like this,
- and are built from the Cube Face Mask and the Batch index

Drawing

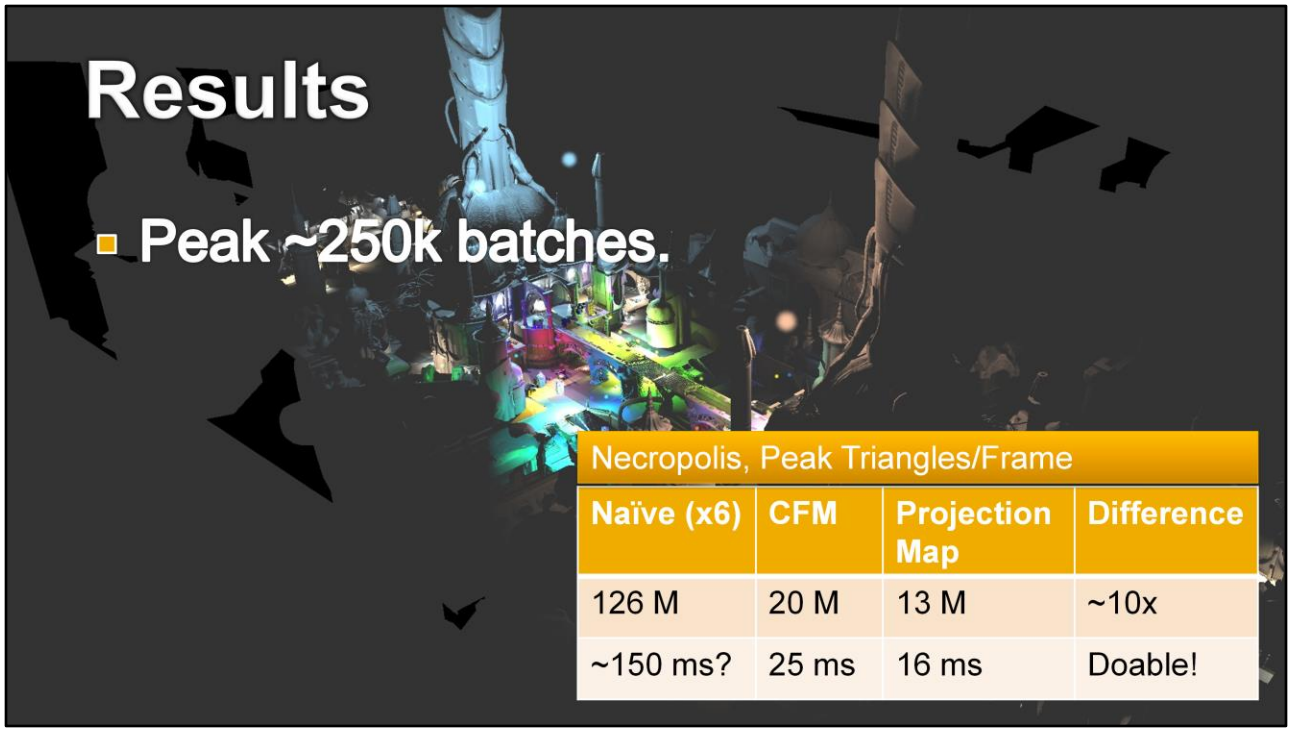
{CFM, batch index}…


```
struct DrawElementsIndirectCommand
{
  uint32_t count = batches[batchIndex].tris * 3;
  uint32_t instanceCount = __popc(CFM); // 0-6
  uint32_t firstIndex = batches[batchIndex].offset * 3;
  uint32_t baseVertex = 0;
  uint32_t baseInstance = index;
};
```

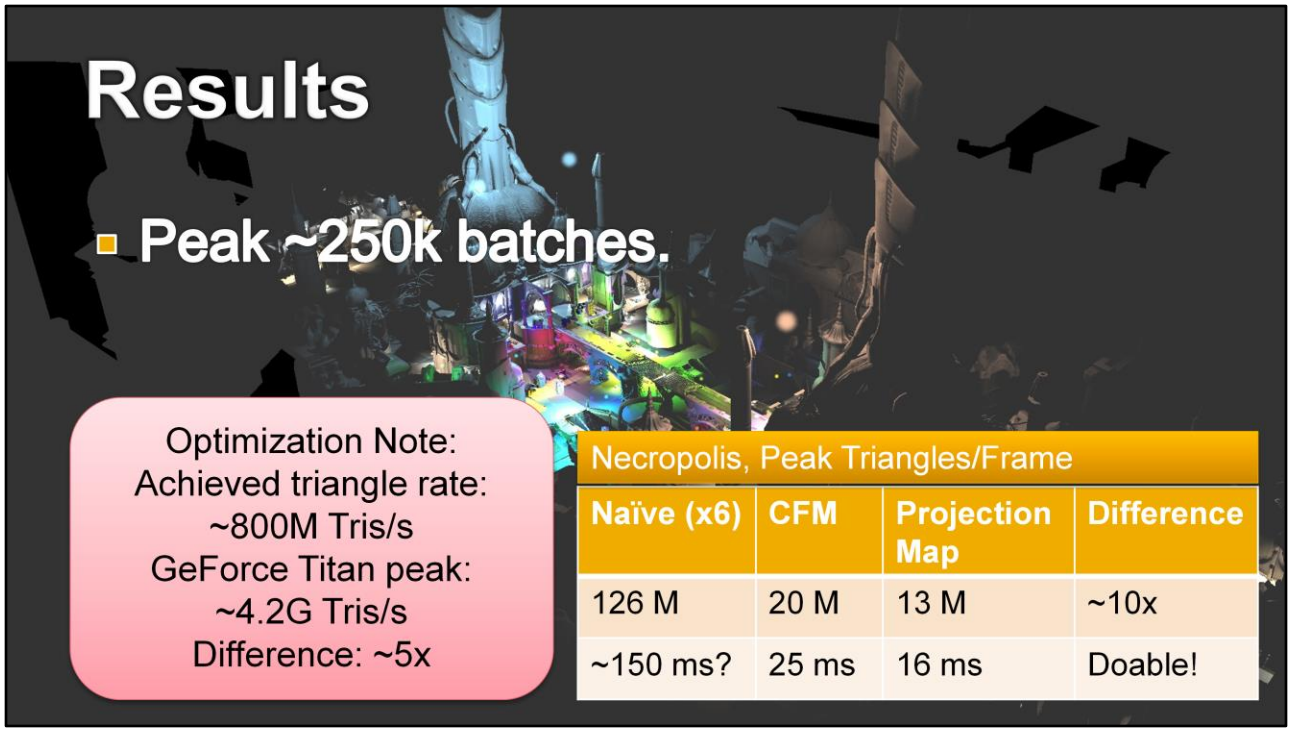
- Note that the number of bits set in the cube face mask correspond to the number of instances drawn
- We use instancing instead of duplication in a geometry shader to draw such batches that overlap multiple cube faces.
- A geometry shader uses the bit mask (through a per-instance attribute) and the instance index to route the batch to the correct cube face.
- The result is very low CPU overhead for the drawing, and good GPU performance.
- When the instance count is zero, nothing is drawn for that command, this is legal OpenGL and we found it to not impact performance much, so we did not find it worthwhile to insert a compaction step to get rid of them.



- At the peak, we draw around 250 thousand batches into the shadow maps, with instancing on top.
- Using the culling techniques just outlined, this results in some 13 million triangles
- which is almost 10 times better than the naïve approach of replicating the triangles to all cube faces.
- And some 65% of performing culling just using the cube face mask.
- The cube face mask path actually already detects completely empty cube faces, and so is already quite effective.



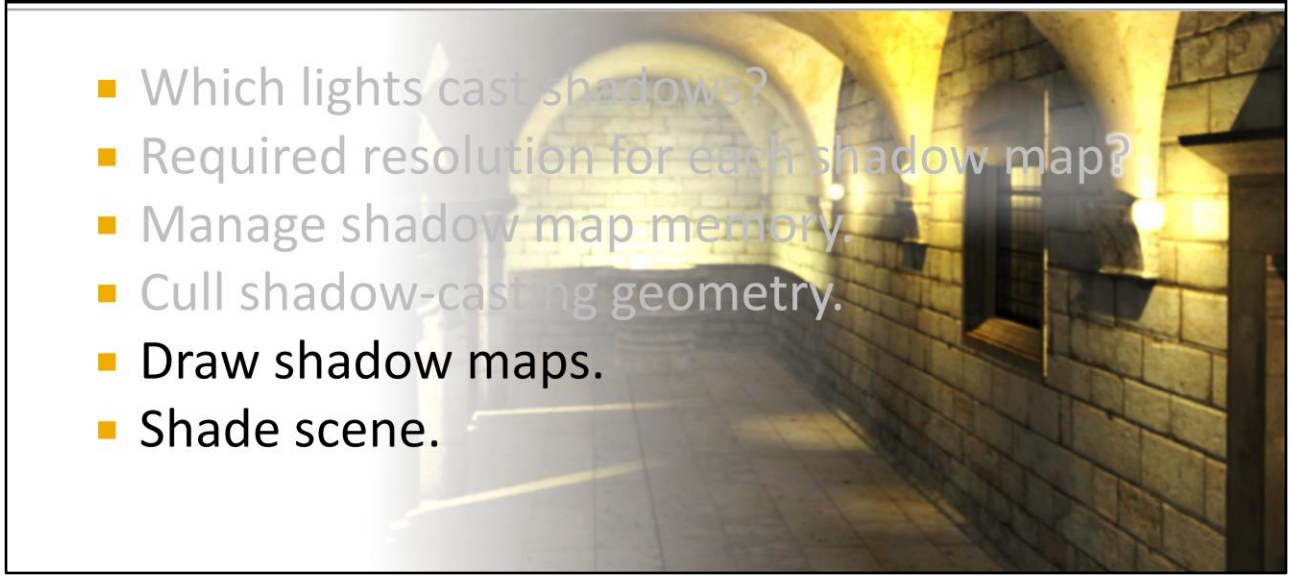
- Again, put differently, this is the difference between infeasible and something we can do in real time.



- As a side note, our implementation achieved a fifth of the theoretical peak triangle rate
- So there could be some pretty good improvements for the handy optimizer out there.

Problem breakdown

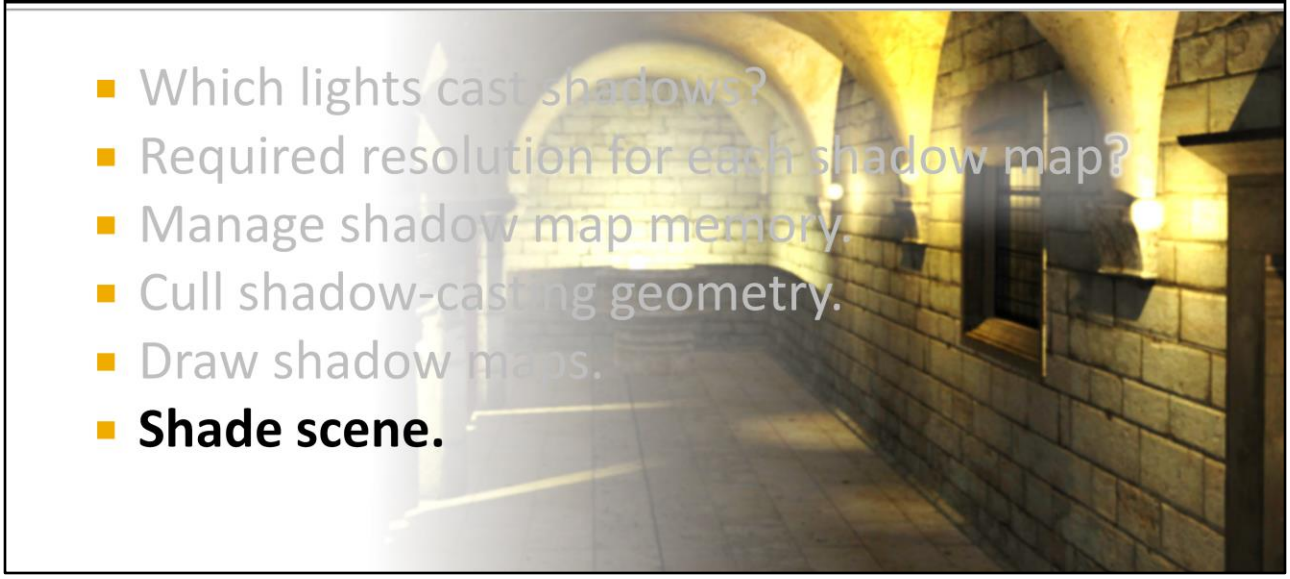
CHALMERS



- We now have all the components needed to shade the scene.

Problem breakdown

CHALMERS



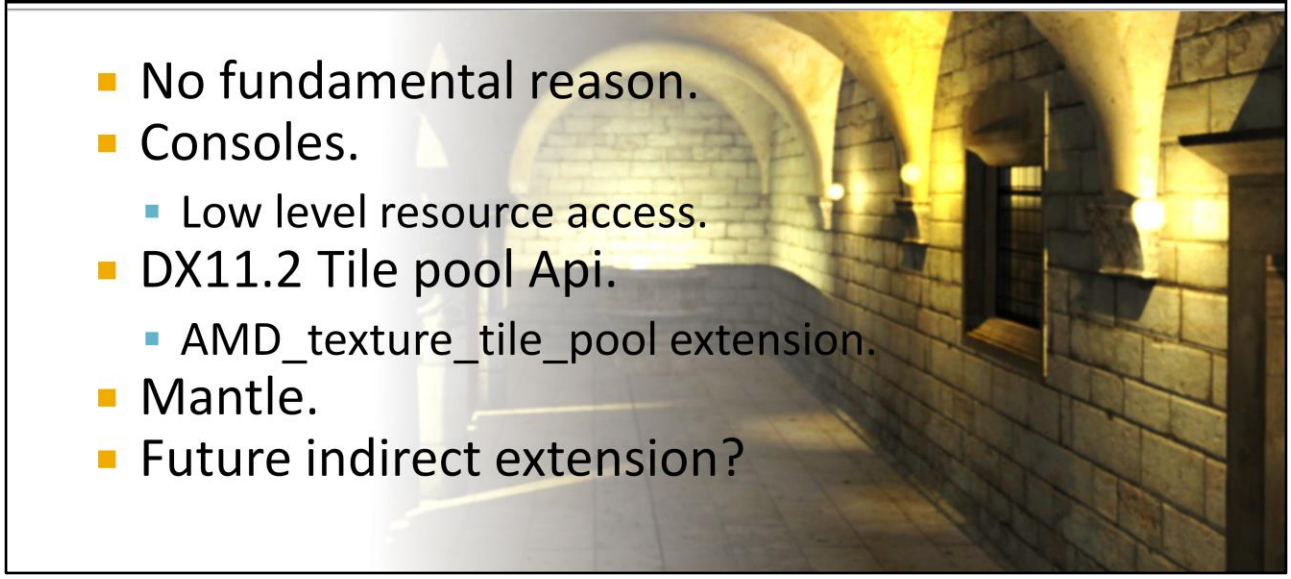
- The shader, be it forward or deferred, just loops over the lights as usual,
- Using the light index it can look up the shadow map (using bindless textures or an array texture or something) and sample it.
- With that, we come to the boring part of the show…

Problem #1

CHALMERS

- Where I have to bring a couple of issues to your attention.
- The first, which is described in the paper (Section 6.3.1 Workarounds), is that page or tile, commits are very slow with the drivers used at the time.
- I have not had opportunity to run again very recently, so things may be better,
- but this is something to keep in mind, if targeting OpenGL on the PC.

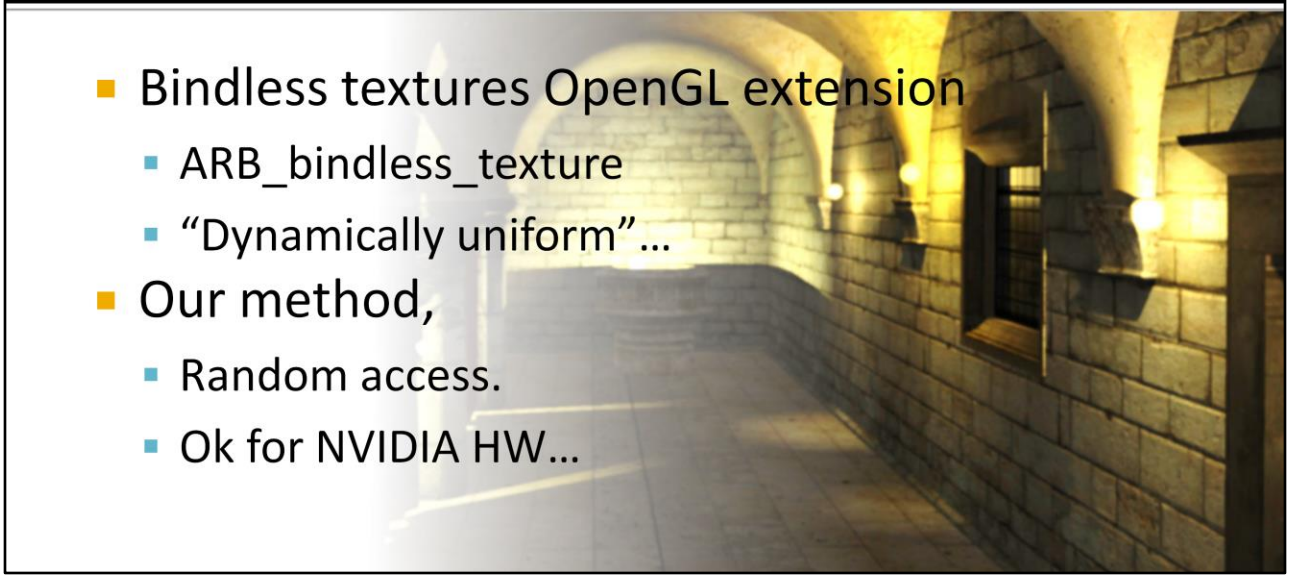
Problem 1 – Slow Commits



- I'm not aware of any fundamental technical reason for the slow behavior,
- So other APIs and platforms may well offer usable performance, you just need to look before you jump here.
- Hopefully there will be extensions in the future to address this on PC.

Problem #2

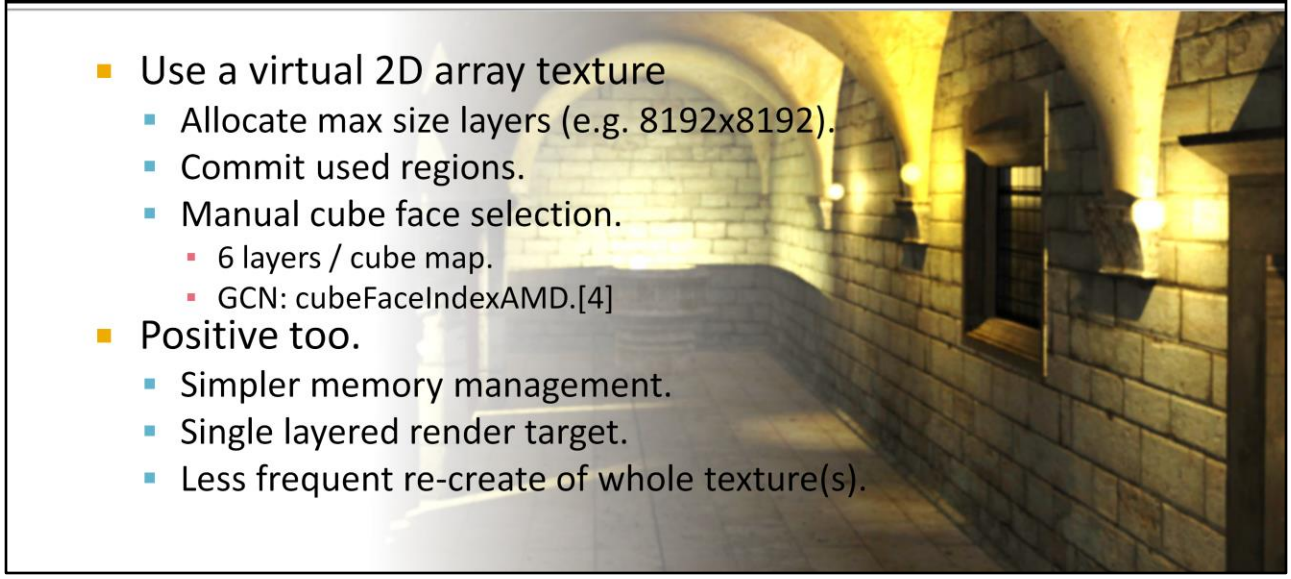
CHALMERS



- The second issue is a compatibility problem.
- In the paper we use bindless textures for the shadow maps.
- …and according to the spec, fragments from the same invocation must use the **same** texture handle.
- Which we absolutely do not!
- This works fine on NVIDIA hardware, but actually doesn’t comply with the spec,

Problem #2 – Bindless

CHALMERS



- Use a virtual 2D array texture
  - Allocate max size layers (e.g. 8192x8192).
  - Commit used regions.
  - Manual cube face selection.
    - 6 layers / cube map.
    - GCN: cubeFaceIndexAMD.[4]
- Positive too.
  - Simpler memory management.
  - Single layered render target.
  - Less frequent re-create of whole texture(s).

- So for other hardware, this can be worked around using a virtual 2D array texture
- But then we must manage cube face selection manually.
- This is not extremely complex and as a benefit, several things actually gets simpler
  - Memory management, as we always just keep the one array.
  - Rasterization, as we can just bind the one layered render target.
  - And it should lead to lower commit/decommit rates.
- This is the way the supplementary demo code is implemented.

Optimization (Low hanging fruit)

- Exploit Temporal Coherency
  - Keep shadow maps between frames.
  - Static light and geometry, and view mostly.
- Static scene parts.
  - No need to recalculate batch AABBs.
  - Batch BVH build can be simplified.
- Simpler Light assignment (Light culling).
  - E.g. Practical Clustered
- Replace 2-pass batch culling with append buffers.

- There are quite a few fairly obvious ways we can improve performance, but which we did not do for the paper.
- The main thing is that we assumed all geometry and all lights were moving, every frame.
- We did this to ensure we were measuring performance for the paper on the most difficult case, as opposed to report at best-case scenario.
- In practice a lot of stuff is going to be static, which means that there is a lot of room to exploit static scene elements, in different ways.
- The most important idea is to simply keep shadow maps between frames, and re-use them if we can detect that the movement is small enough. For completely static lights, with no moving geometry this should be quite simple, but it is not completely clear how to detect. One heuristic that is often used anyway, is that lights far away can just be updated less often.
- We reused our hierarchical light assignment code from previous papers, but it is really dimensioned for huge numbers of lights and much simpler code would be much faster (e.g., Emils design).
- There are also more low level opportunities.

Further reading.

[1] My own publications and demos. http://www.cse.chalmers.se/~olaolss

[2] Forward+ related info, Harada et al. http://scholar.google.com/citations?user=4R7xOcsAAAAJ&hl=en

[3] Karras et Aila, HPG 2013, *Fast Parallel Construction of High-Quality Bounding Volume Hierarchies*

[4] OpenGL extension registry, `ARB_sparse_texture`, `ARB_bindless_texture`, `AMD_gcn_shader`

[5] Coombes, GDC 2014, *Taking Advantage of DirectX11.2 Tiled Resources*

[6] Forsyth, GDC 2004, Multiple Shadow frustra http://home.comcast.net/~tom_forsyth/papers/shadowbuffer_pseudocode.html

[7] HOLLANDER, M., RITSCHEL, T., EISEMANN, E., AND BOUBEKEUR, T. 2011. ManyLoDs: parallel many-view levelof- detail selection for real-time global illumination. Computer Graphics Forum 30, 4, 1233–1240.

[8] Wolfgang Stürzlinger and Rui Bastos. Interactive rendering of globally illuminated glossy scenes. In Proc. of the Eurographics Workshop on Rendering Techniques '97, pages 93–102. Springer-Verlag, 1997.

[9] Carsten Dachsbacher and Marc Stamminger. Splatting indirect illumination. In Proc. I3D '06, page 93–100. ACM, 2006.

[10] LEFOHN, A. E., SENGUPTA, S., AND OWENS, J. D. 2007. Resolution-matched shadow maps. ACM Trans. Graph. 26, 4 (Oct.).

Many Light Rendering on Mobile Hardware

Part 4/4 - 40 min

Presenter: Markus Billeter
Chalmers University of Technology

Overview

- Introduction - Mobile Hardware
  - What's different?
  - Why do we need to treat it separately?
- Different methods
  - Properties – focus on Mobile Arch.
  - Somewhat repetition from Ola's Introduction
- Practical Clustered Forward Implementation

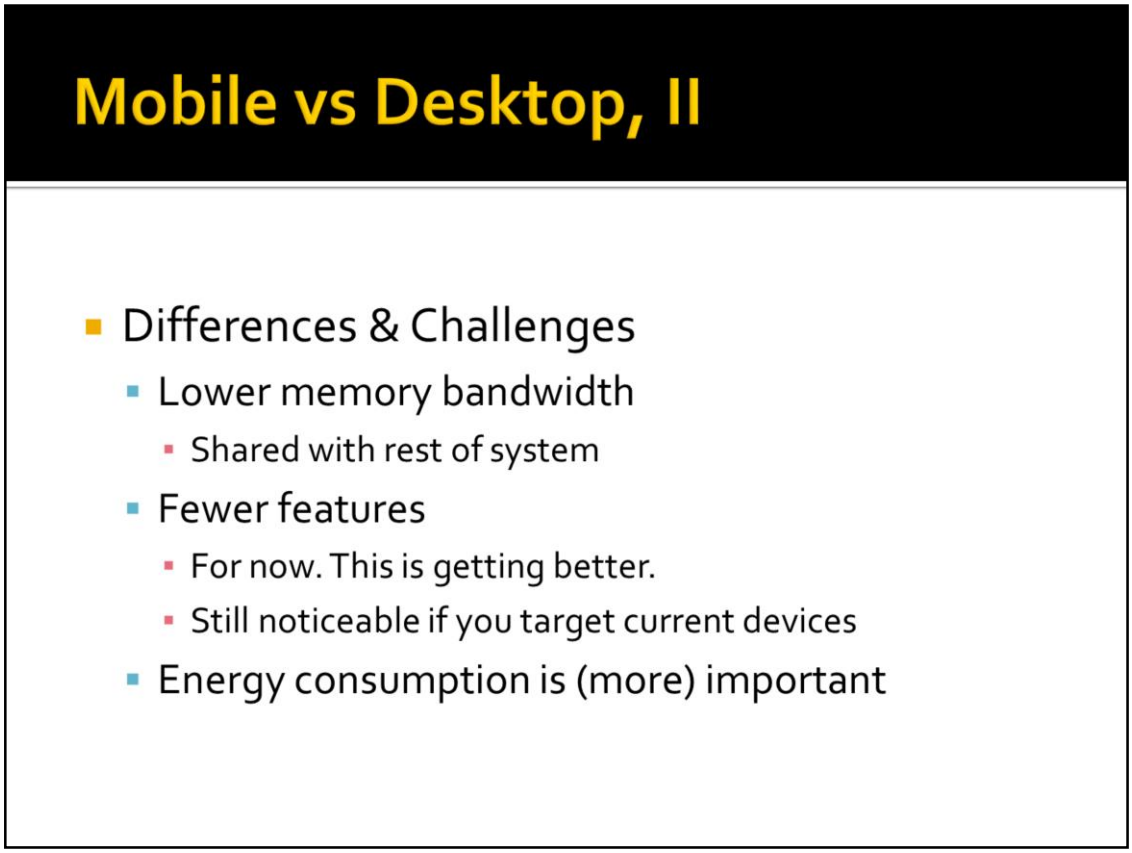
The presentation is divided into these three parts. I'm first going to talk about mobile hardware, and motivate why it deserves special consideration when picking a rendering method. After that, I'm going to revisit different many-light rendering methods, although with a focus on their properties with respect to mobile hardware requirements and limitations. Some of this will be a repetition from Ola's introduction at the very beginning of the course – but there are also two new methods that I'll mention. Finally, I'm going to talk about the method I ended up picking for doing many-light rendering on Android-class hardware.

Introduction – Mobile Hardware

Mobile vs Desktop

- So far considered modern high-end systems
- How is mobile hardware different?
  - What are the challenges?

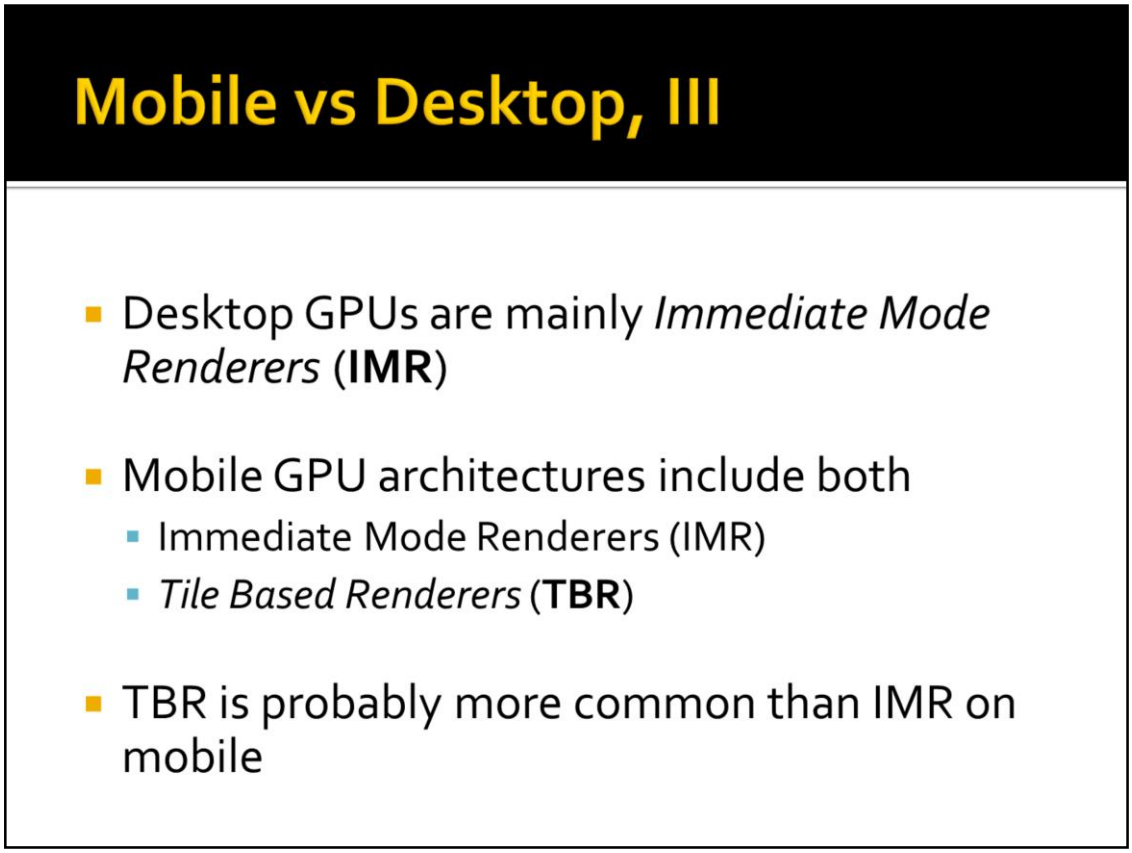
So far, the course has mainly considered high-end systems – i.e., desktop-class GPUs and perhaps consoles (Emil's part). Now that we're looking at mobile hardware, we need to find out how it differs and what kind of challenges it poses.



One of the main differences is that most mobile hardware will have much lower memory bandwidth when compared to desktop systems. The memory bandwidth available to the GPU may also be shared with the rest of the system, since both GPU and CPU share the same memory (unlike dedicated desktop GPUs, which have separate VRAM).

For now, mobile GPUs still have fewer features, when compared to a modern desktop GPU. This might be getting better with e.g., newer OpenGL|ES versions, but if you're targeting current mobile devices you will notice quite some limitations.

Pretty much every talk on mobile rendering mentions energy consumption. If we can do something on the software side to reduce energy consumption, that's certainly worth considering. On the other hand, this is perhaps one of the more difficult items from a software perspective, especially if we consider the very different architectures that exist.



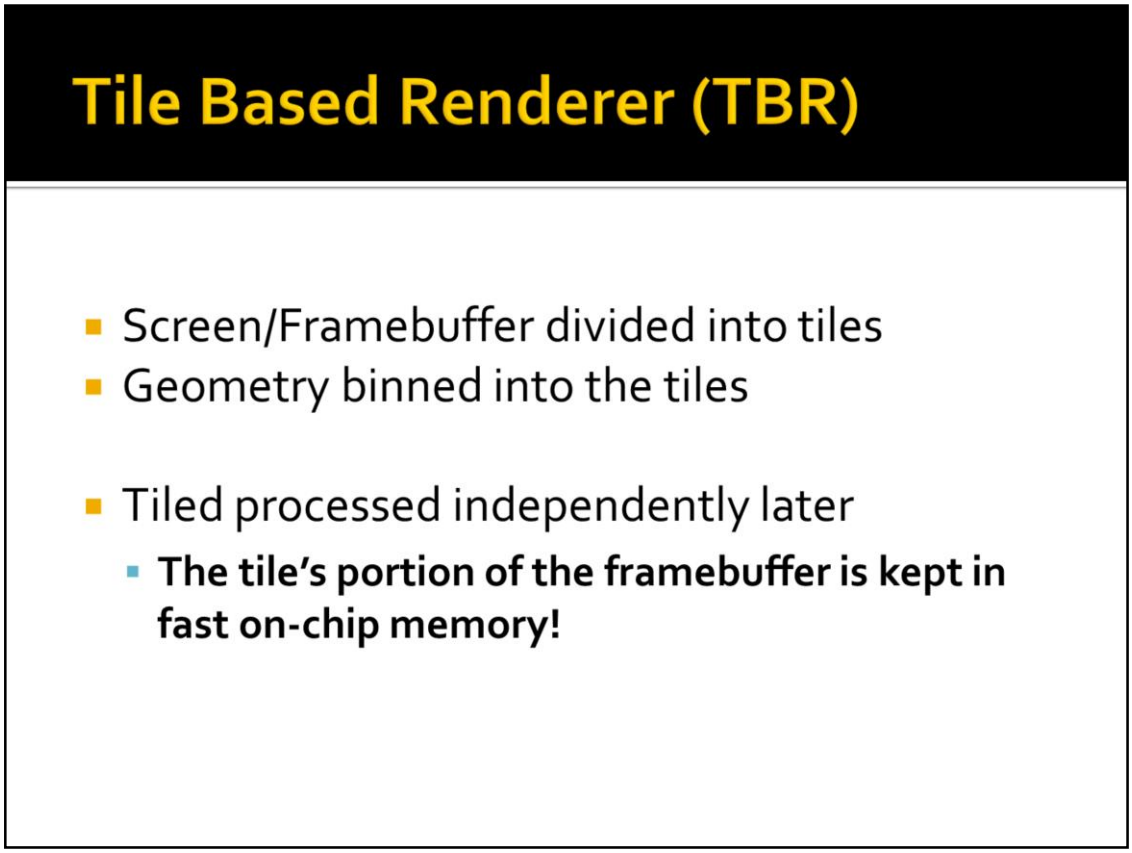
On the topic of different architectures. If we look at desktop class GPUs, we're mostly looking at immediate mode renderers (IMR). What I mean by this, I'll explain on the next slide.

Mobile GPUs include both immediate mode renderers and tile based renderers (TBR) – again, I'll tell you what a TBR is in a couple of slides. The important point is that both of these architectures commonly occur – in fact, TBR is probably more common than IMR when considering mobile platforms.

Immediate Mode Renderer (IMR)

- "Normal" rendering
- Geometry is transformed and then rasterized immediately
- Framebuffer typically resides in VRAM
  - VRAM is written to multiple times

An immediate mode renderer is what I'd consider "normal rendering", i.e., as typically shown when illustrating e.g. the OpenGL pipeline. The program submits geometry to the GPU in batches, and the geometry is transformed and rasterized as it is received by the GPU. The framebuffer in an IMR architecture typically resides in VRAM in its entirety. The VRAM might be written to multiple times when there's overdraw.

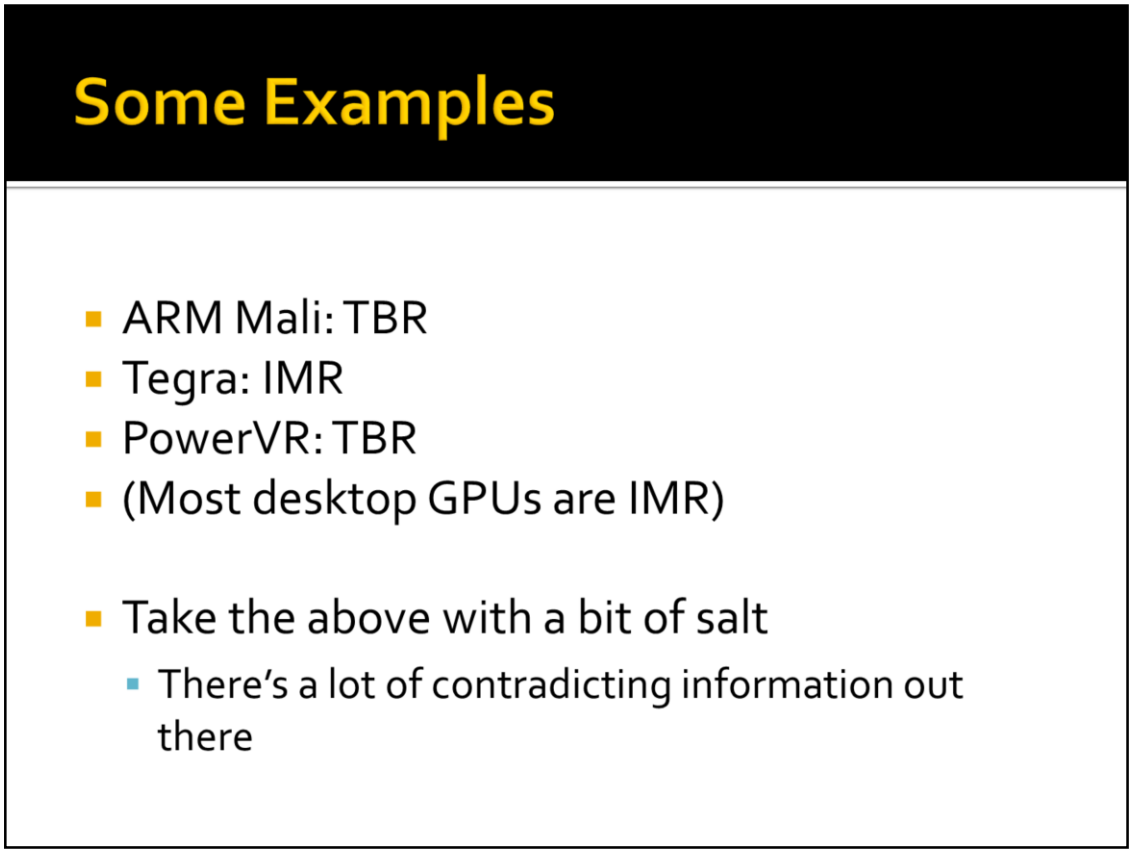


On a tile based renderer, the screen/framebuffer is divided into tiles – which gives this architecture its name. When the application submits geometry, it is binned into these tiles (rather than rasterized "immediately"). Later – for example when all geometry has been submitted – the tiles are processed independently. Each tile's portion of the framebuffer is kept in local on-chip memory, thereby avoiding expensive writes to RAM.

Tile Based Renderer (TBR), II

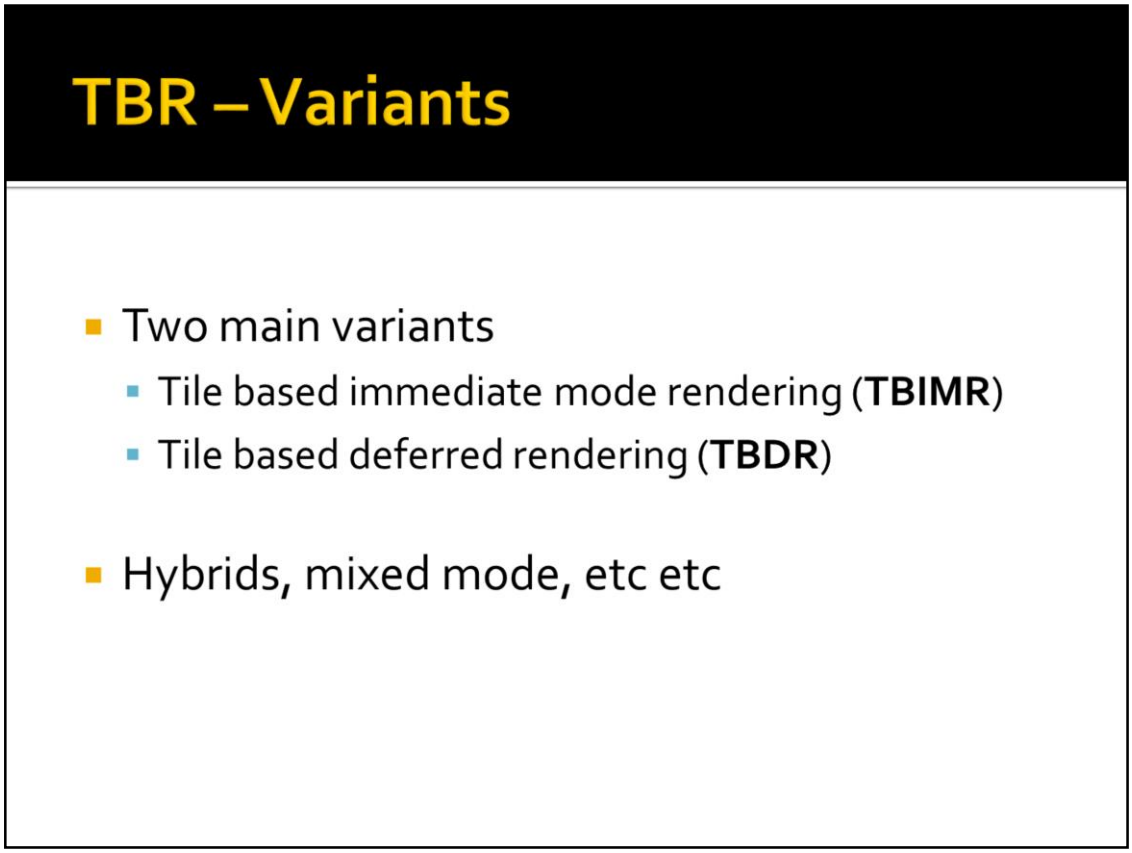
- Tile's contents stored to RAM when needed.
- In the best case:
  - Once, when **all** rendering to that tile has finished

Only when needed the tile's contents are stored to RAM. In the best case this occurs only once every frame, when all the rendering for that frame has finished.



Some examples of mobile architectures. As you can see, both TBR and IMR are present in this list.

Please also take this with a bit of salt, there's quite a bit of contracting information out there regarding what architecture different GPUs have. If you know better here, feel free to correct me.

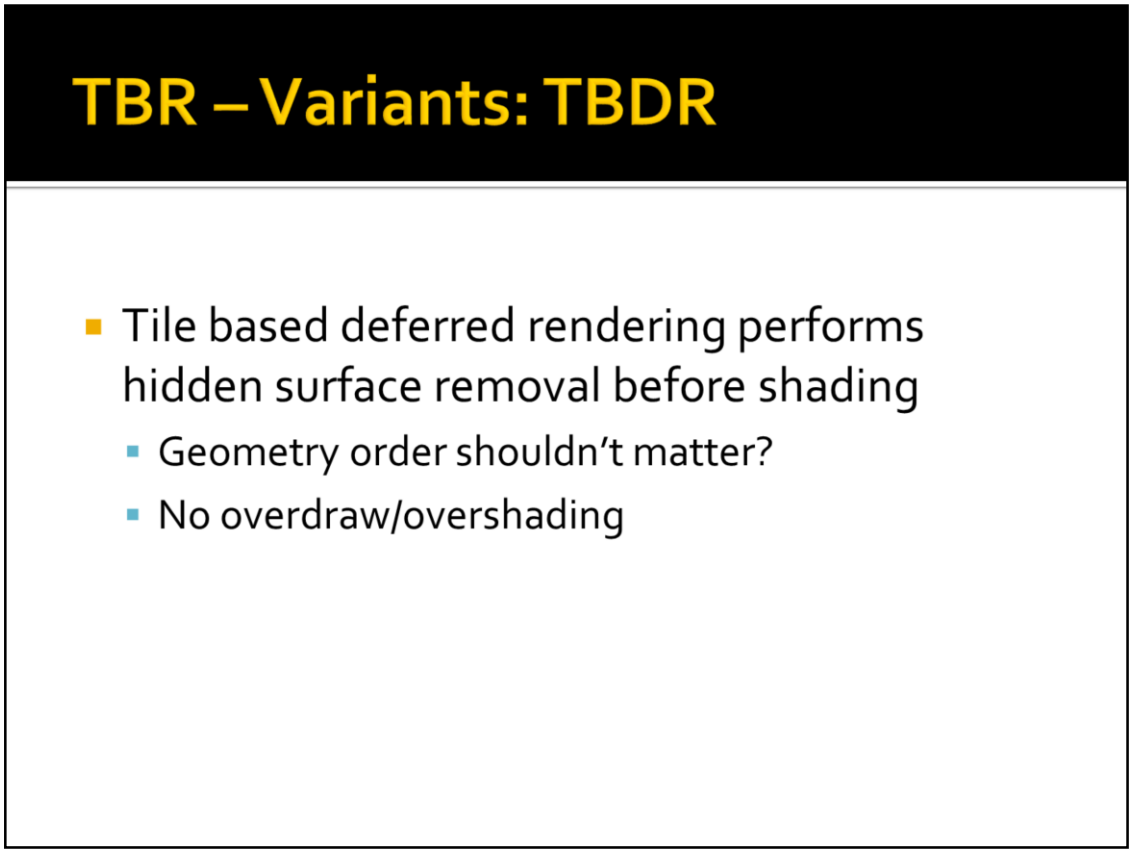


TBR additionally comes in the variants: tile based immediate mode rendering, and tile based deferred rendering. Besides this, there seem to be GPUs that can switch between e.g., TBR and normal IMR, various hybrids etc etc. But let's quickly look at TBIMR and TBDR.

TBR – Variants: TBIMR

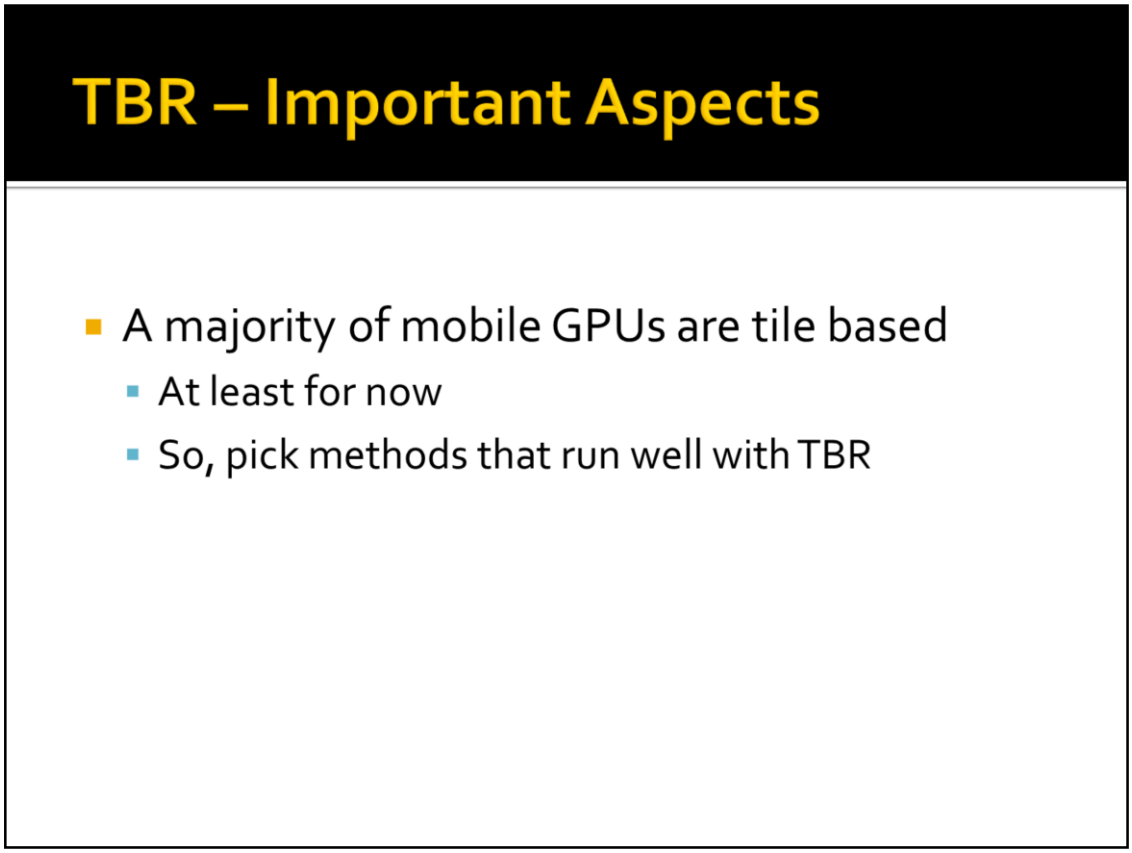
- Tile based immediate mode rendering rasterizes per-tile contents similar to IMR
  - I.e., uses a depth buffer with e.g. EarlyZ to do hidden surface removal
  - Submit geometry front-to-back
  - Overdraw can occur

In TBIMR, the geometry binned into each tile is rasterized in a fashion similar to IMR when processing a tile. That is, the geometry is processed in the order it was submitted from the application. The tile has a depth buffer, typically with support for EarlyZ, to perform hidden surface removal. But because the geometry is processed and shaded in the order of submission, overdraw and overshading can occur. This can be mitigated somewhat by e.g., submitting the geometry front-to-back.



In TBDR, the hidden surface removal is performed before shading. This means that geometry order shouldn't matter, and that there should be no overdraw/overshading.

TBDR is apparently employed by PowerVR GPUs. Unfortunately I've not had the chance to test this on PowerVR GPUs myself, so I can't verify that this actually is the case.



Anyway, either way, a majority of the mobile GPUs are tile based at the moment. So, we really want to pick a method that performs well on a TBR.

TBR – Important Aspects, II

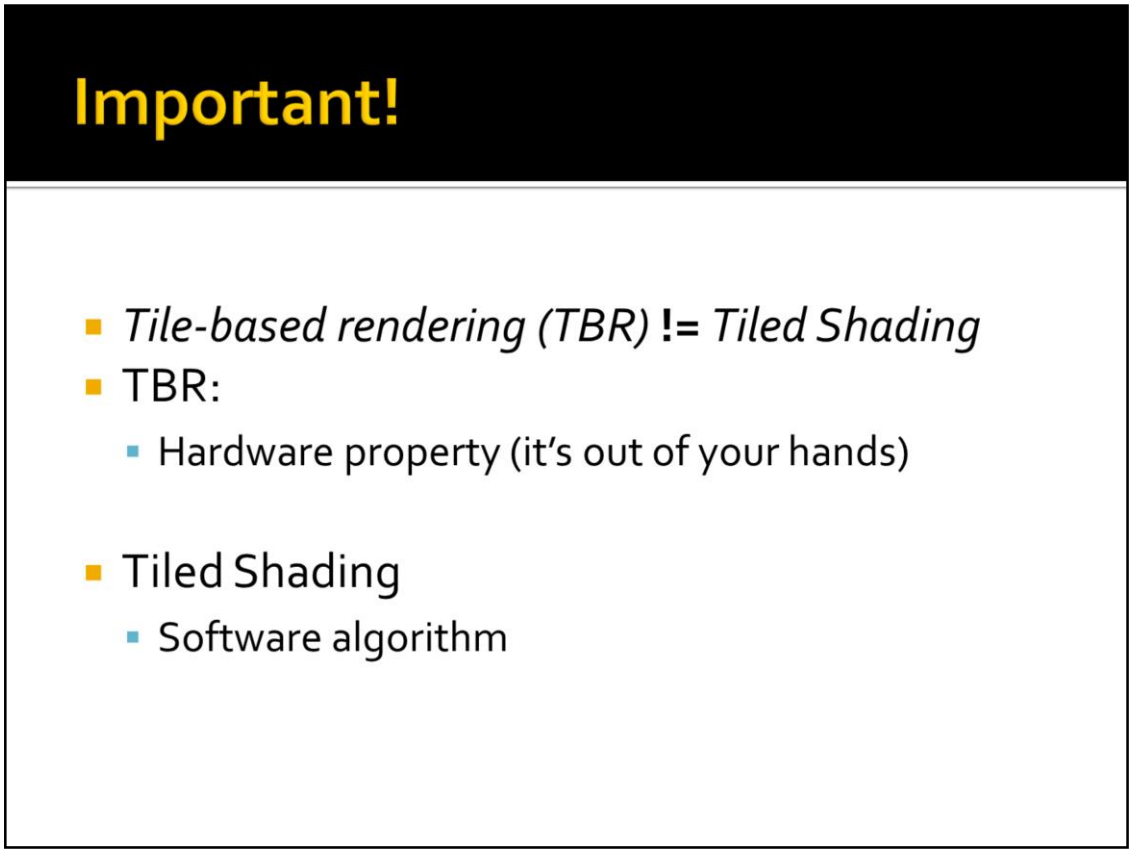
- All TBR keep the tile's framebuffer contents in **fast** on-chip memory
  - We want to make sure the data stays there
  - Loads/stores to/from RAM use precious memory bandwidth
    - With all the implications to performance and energy consumption

All TBR architectures keep the tile's framebuffer contents in fast on-chip memory. We want to make sure the data stays there, since loads and stores to and from RAM use precious memory bandwidth. Not only is the bandwidth limited, but it apparently is expensive in terms of energy consumption as well.

TBR – Important Aspects, III

- So, on a tile based renderer we want to keep data in the on-chip buffer when possible
  - True for both TBR variants (TBIMR and TBDR)

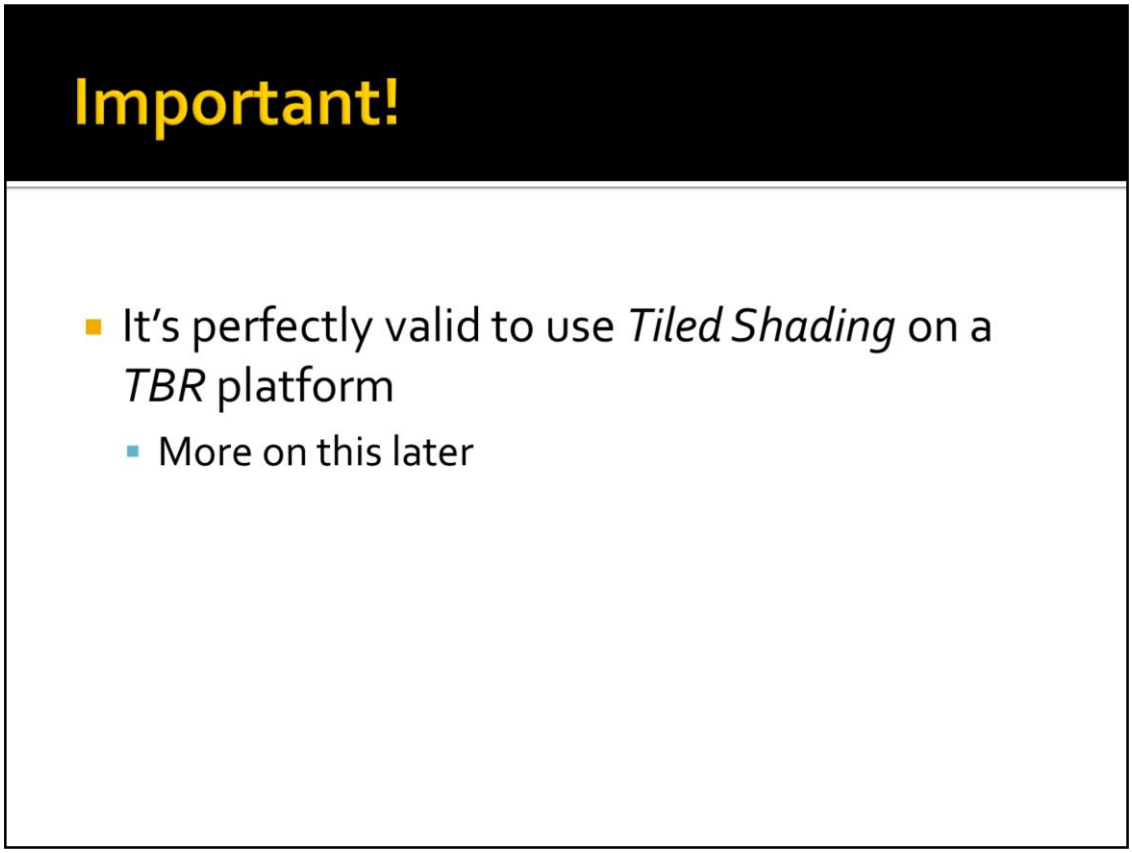
So, again, on a TBR we want to keep the framebuffer data on-chip whenever possible. This is true for both TBR variants, i.e., it holds for both TBIMR and TBDR.



Short note: tile based rendering – the thing I've been talking about so far is completely unrelated to tiled shading (the thing that Ola was talking about in his introduction).

TBR is a hardware property, so it's largely out of your hands – unless, let's say, you flat-out refuse run on TBR platforms.

Tiled shading on the other hand is a software algorithm that you can pick.



In fact, it's perfectly valid to use tiled shading (the software algorithm) on a tile based rendering platform. It might even be a good idea, as we'll see later.

An other aspect of mobile devices is that we have to use slightly different APIs when dealing with tem. On the desktop we have both “normal” OpenGL and OpenGL|ES. On mobile devices, we’re typically limited to OpenGL|ES.

OpenGL|ES comes in mainly two versions: 2.0 and 3.x. There’s also 1.x (obviously), but that’s rather uninteresting for our purposes.

OpenGL|ES, II

- OpenGL|ES 2.0
  - Supports vertex + fragment shader
  - Has framebuffer objects, but doesn't support multiple render targets (**MRT**)
  - Limited selection of texture formats
  - Limited selection of renderable formats

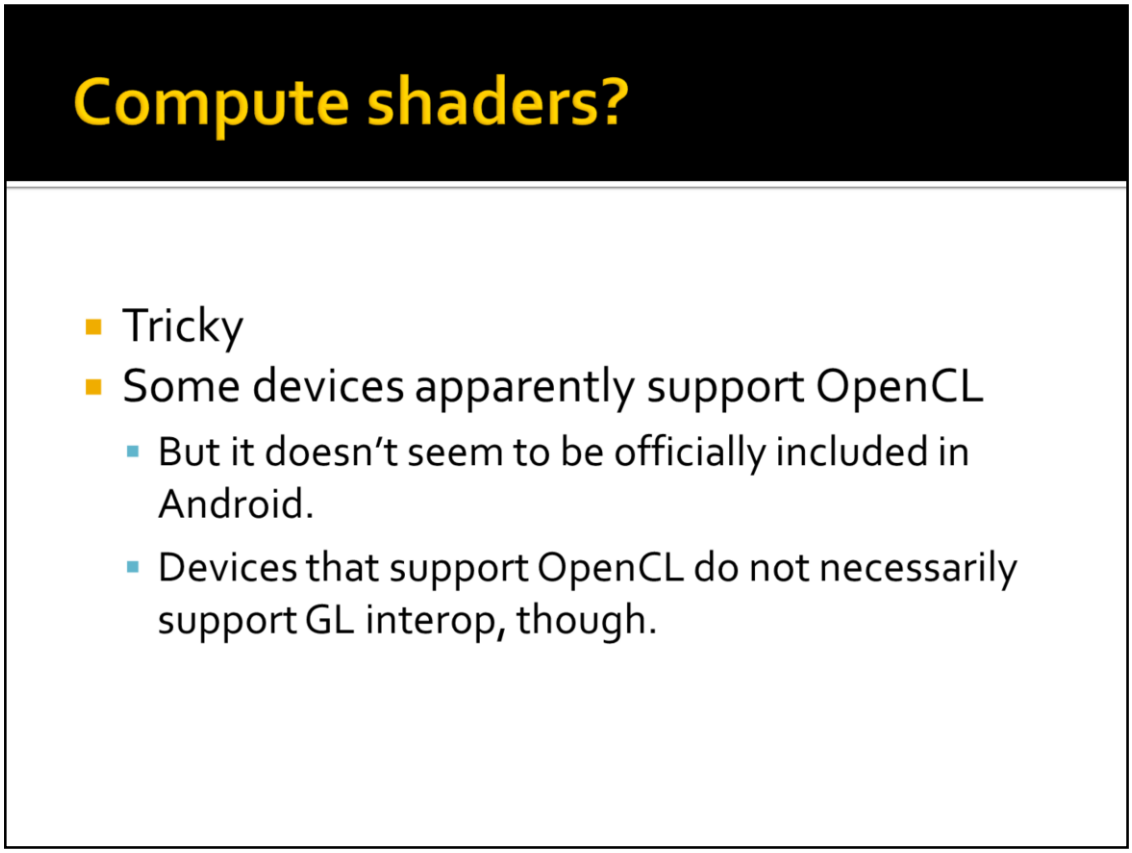
OpenGL|ES 2.0 is sort-of-modern, in the sense that it supports vertex and fragment shaders. It has framebuffer objects, but in vanilla OpenGL|ES 2.0, these are limited to a single color attachment. So there's no support for multiple render targets.

The available formats for textures and framebuffer attachments are also somewhat limited. None of the core formats have more than 32 bits per pixel, and you might not even be guaranteed this without extensions.

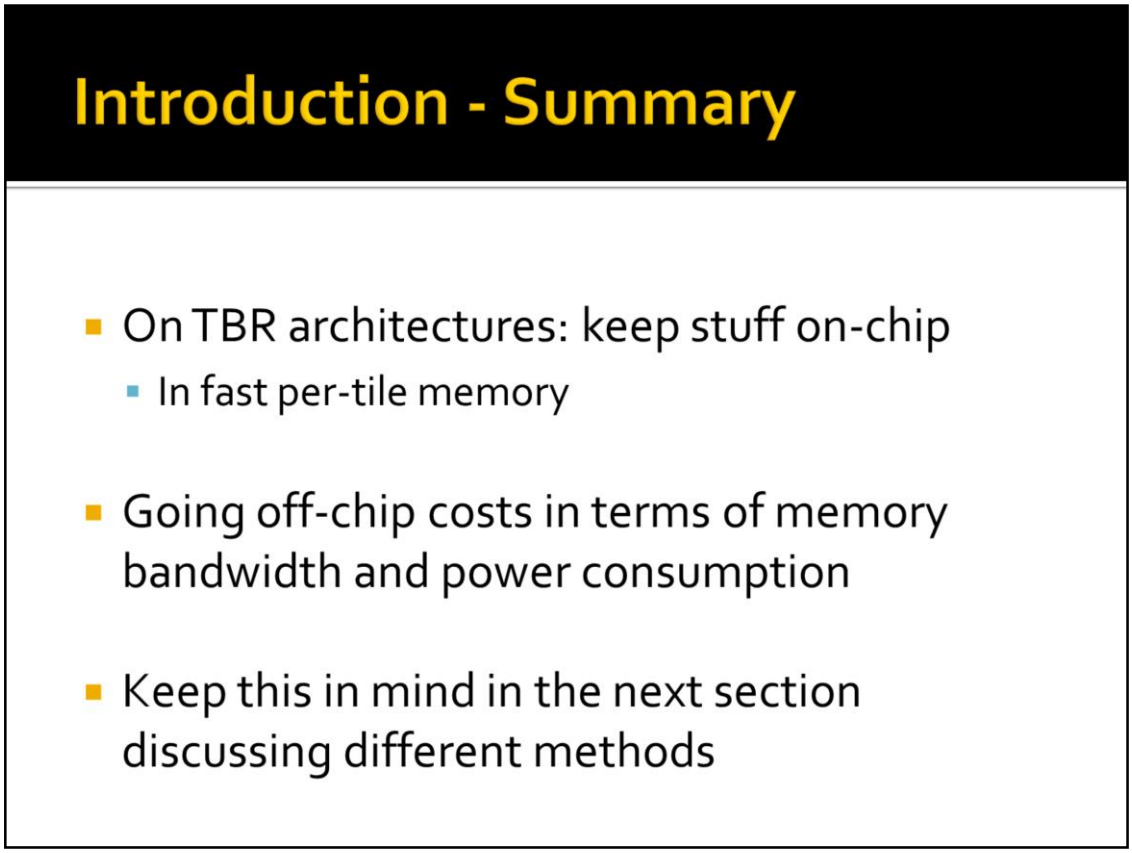
OpenGL|ES, III

- OpenGL|ES 3.0
  - Improves many aspects
  - Example: support for MRT, transform feedback etc.
  - Actually available on various Android devices.

OpenGL|ES 3.0 improves the situation in many ways. For example, it supports multiple render targets, has transform feedback etc etc. And it's actually available on various Android devices these days.



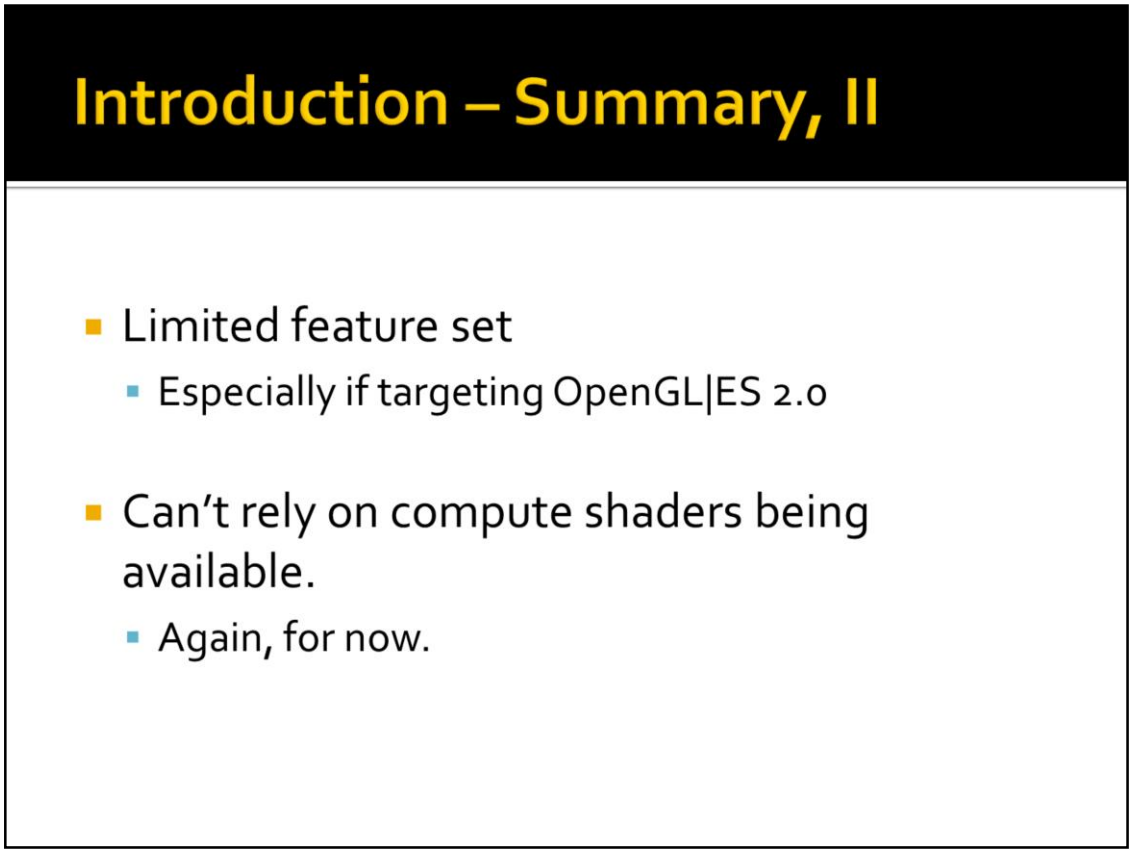
However, the situation still looks a bit dark when trying to do GPU-compute tasks. Some Android devices support OpenCL, but OpenCL doesn't seem to be officially included in the android ecosystem. Also, when playing around with OpenCL on the Nexus 10, I noticed that there was no support for interop functions between OpenGL|ES and OpenCL at the time.



So, let's quickly review what I've said so far.

On TBR architectures, which are very common in the mobile world, we want to keep the data on-chip, in fast per-tile memory. Going off-chip uses memory bandwidth, which costs both in terms of performance and power consumption.

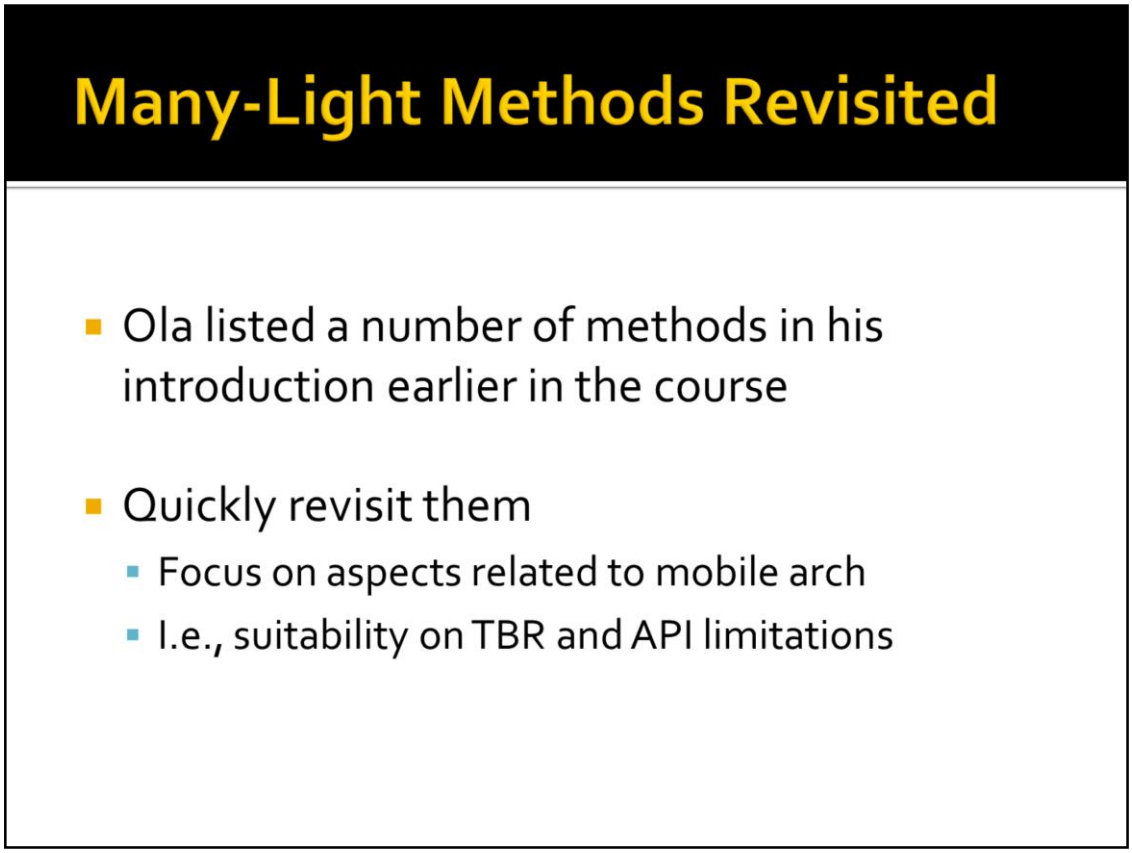
When discussing the different methods in the next section, we'll keep this in mind.



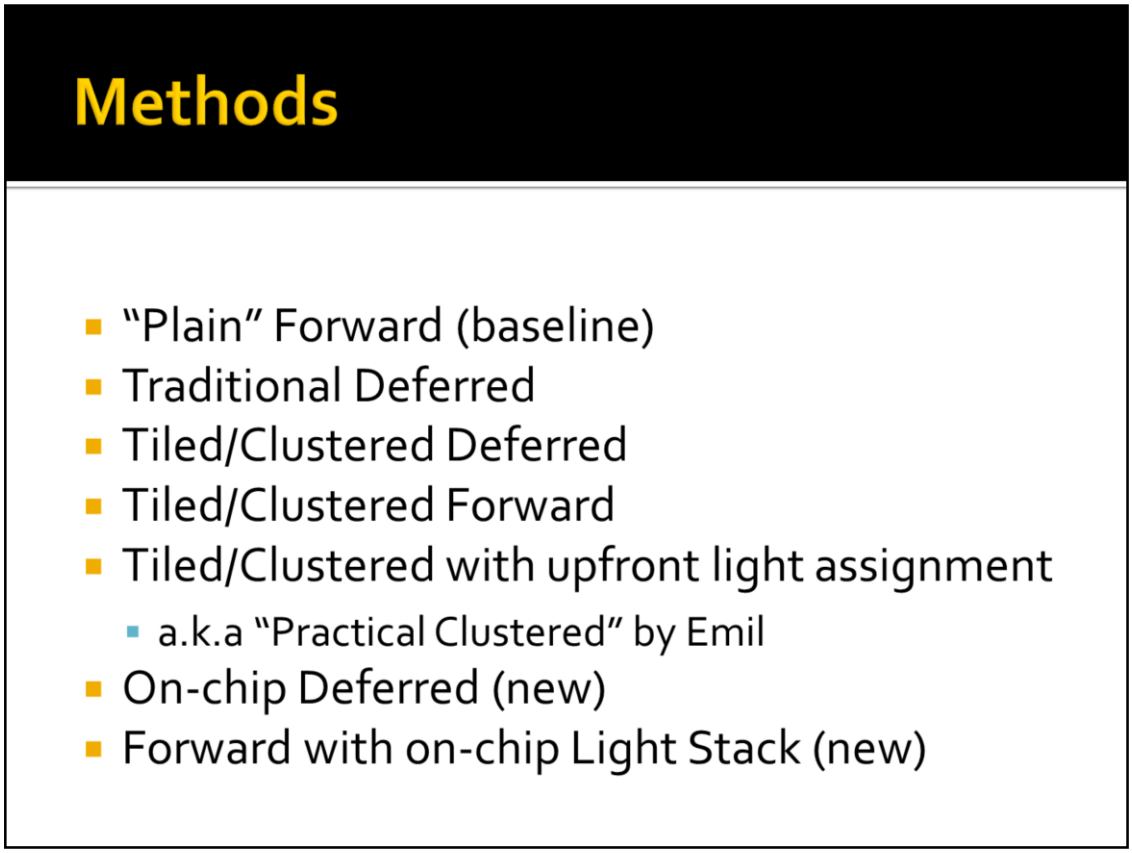
Mobile GPUs still have a somewhat limited feature set – although this is rapidly improving. It's still an issue if you want to target OpenGL|ES 2.0 devices, though.

I wouldn't at the moment rely on compute shaders being available, at least not if targeting Android tablets and/or phones.

Many Light Methods Revisited

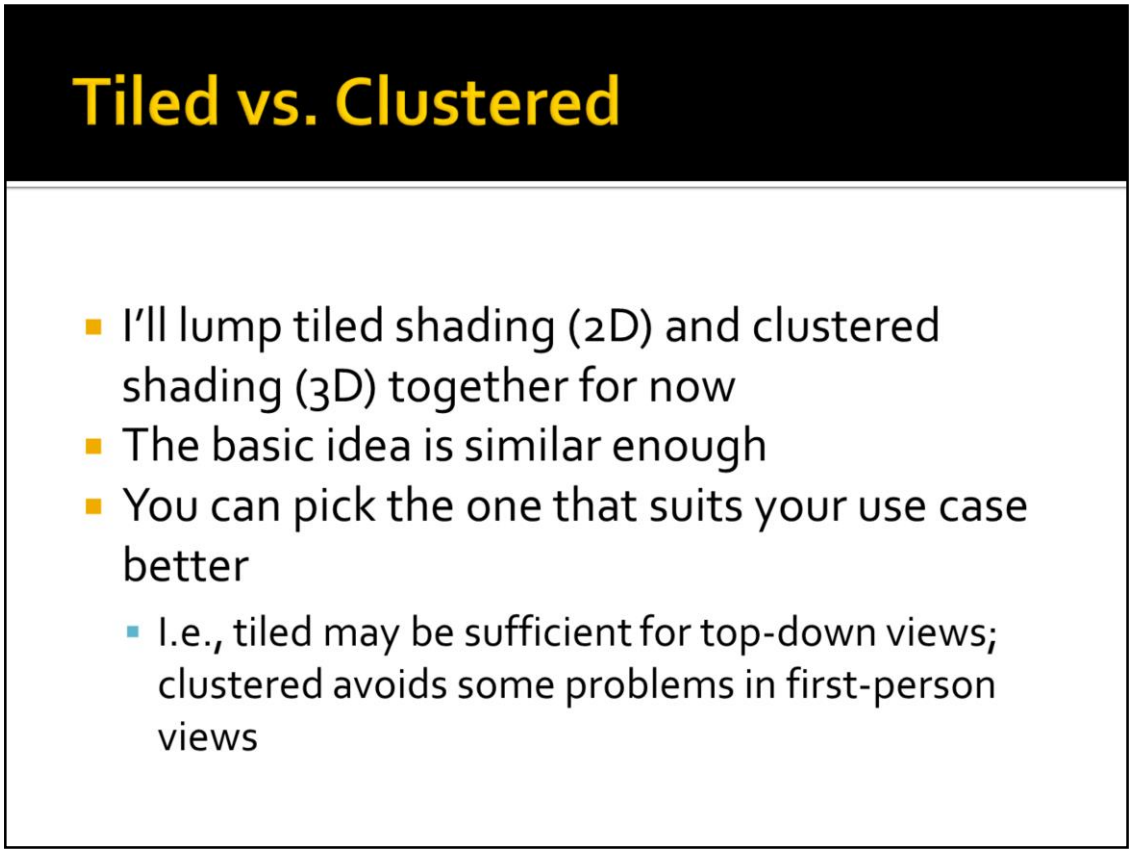


In the beginning of this course, Ola listed and described a number of many-light rendering methods. I'll quickly revisit some of them, but this time focusing somewhat on the method's properties relating to mobile platforms. In particular, the method's suitability for TBR platforms is interesting.

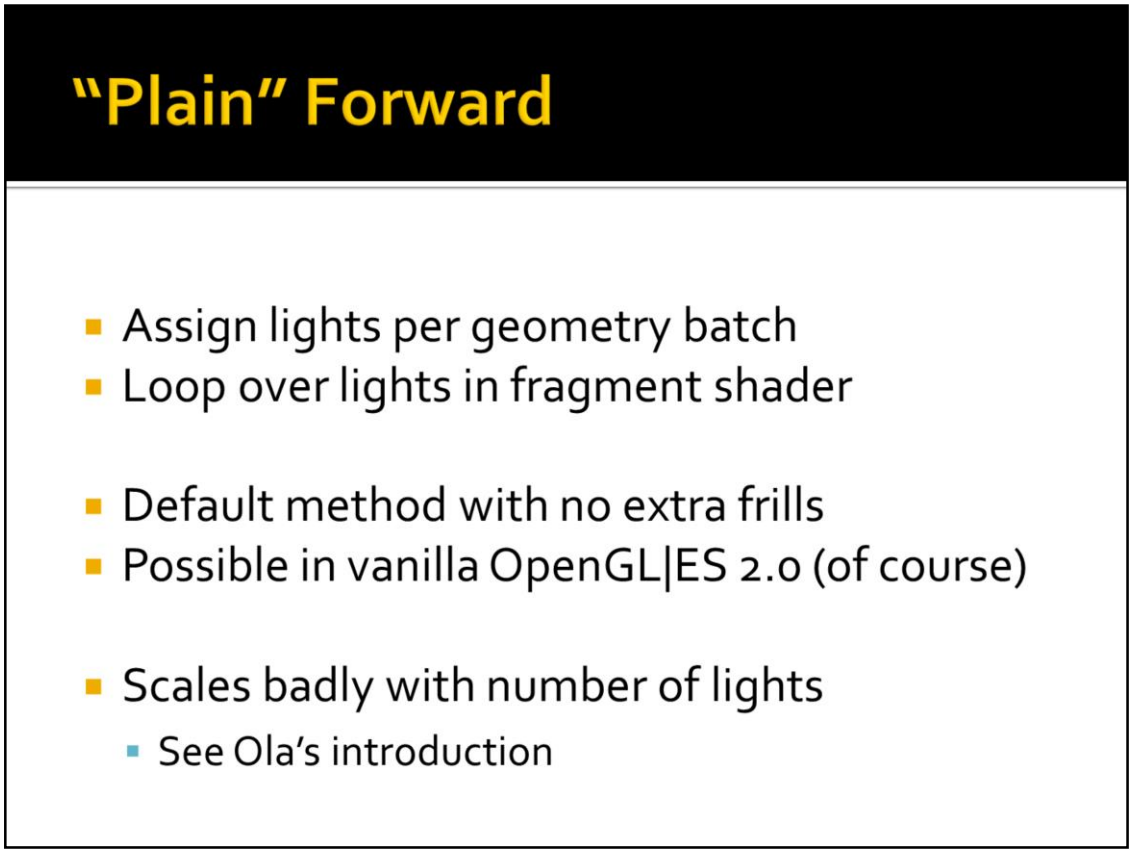


The methods that I'm going to cover are listed here.

The first method – plain forward – is included for reference, I wouldn't call it a many-light rendering method as such. Also, the last two methods are new, and are constructed with TBR platforms in mind. They utilize extensions that enable them to take further advantage of the tile on-chip memory. Both were previously presented by Martin et al. at e.g., SIGGRAPH 2013.



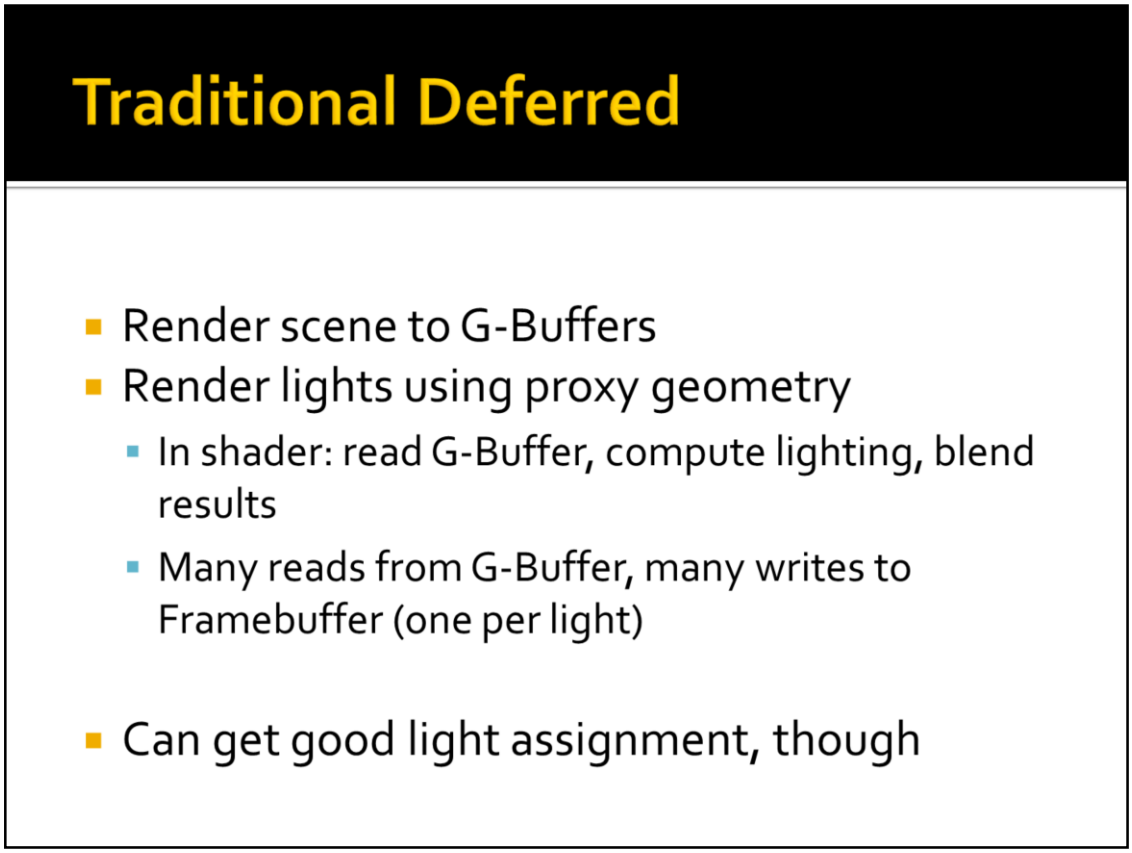
During this overview, I won't distinguish between tiled and clustered too much. Both are very similar – in fact, you could consider tiling to be a special case of clustering with just a single depth slice. I think this is a valuable view anyway – you can adapt the way clustering is performed to match your problem. The extreme cases of this are pretty much tiled on one hand, and the original clustered method on the other hand. So, using tiled shading if you mainly have top-down views with little depth complexity is a perfectly good choice. If you have more complex views, clustered might be a better choice, since it avoids some problems with e.g., depth discontinuities, as previously discussed in the course.



Anyway.

The first method here is the "plain" forward rendering. This is pretty much what you get when you just do text-book OpenGL rendering, that is, you submit all your geometry in batches. You can assign lights to individual batches; in the fragment shader, you'll then simply look over all lights.

Since this is pretty much the default way of doing OpenGL rendering, it's obviously possible in OpenGL|ES 2.0 without any extensions. It's not really a many-light rendering method, though, since, as explained by Ola in the introduction, assigning lights to geometry batches may scale rather badly with increasing number of (dynamic) lights.



Next up, there's the traditional deferred shading. To quickly recap: here, you render the scene to G-Buffers that store the geometry's information required for shading. This includes at the very least an albedo/color, a normal and a position (although the latter can be reconstructed from the depth value).

After rendering the geometry into the G-Buffer, lights are splatted by rendering their bounding volumes. In the fragment shader for this step, the G-Buffer is read, lighting computed and the results are blended into the framebuffer. Each light that touches a certain pixel will require one read from the G-Buffer and one write to the framebuffer. This can use up quite some memory bandwidth, which is also the reason people started looking at Tiled Deferred shading and similar techniques.

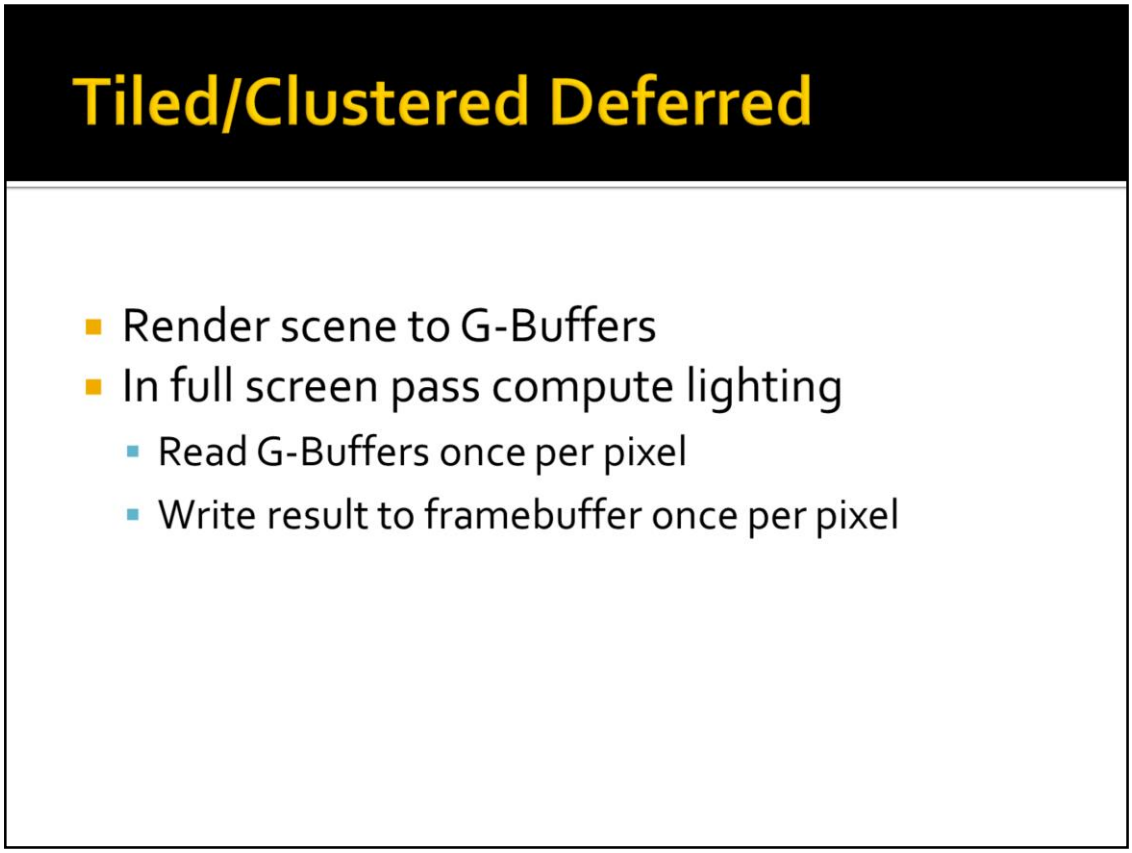
It's worth nothing that you can get very good light assignment with traditional deferred, though – in some cases much better than with tiling and even clustering.

Traditional Deferred, II

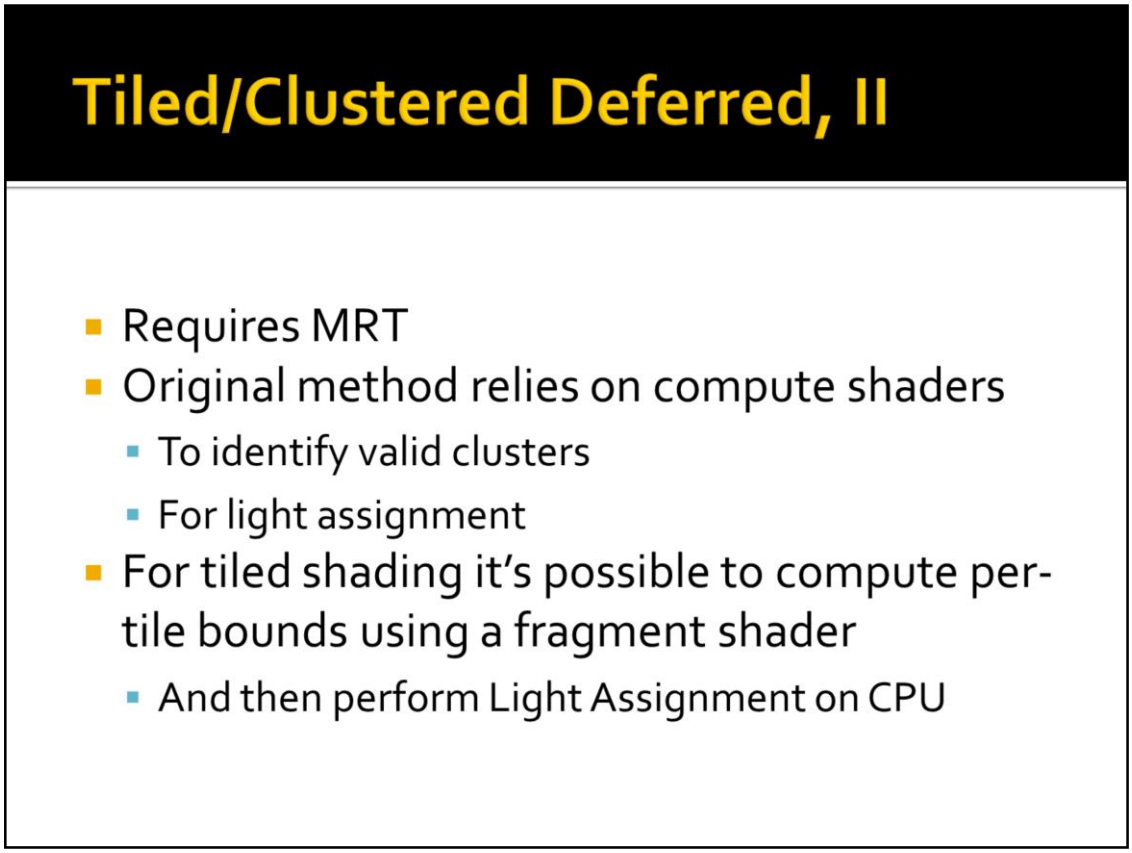
- Requires multiple render targets
  - I.e., not supported by vanilla OpenGL|ES 2.0
- G-Buffers are expensive in terms of memory bandwidth
  - Copied off-chip, so they can be accessed through textures

Traditional deferred requires support for multiple render targets – or you'll end up doing really lots of geometry passes. As such, it's not supported by vanilla OpenGL|ES 2.0.

Also, G-Buffers are expensive in terms of memory bandwidth even on desktop GPUs, so memory bandwidth is definitively an issue on mobile hardware with deferred methods.



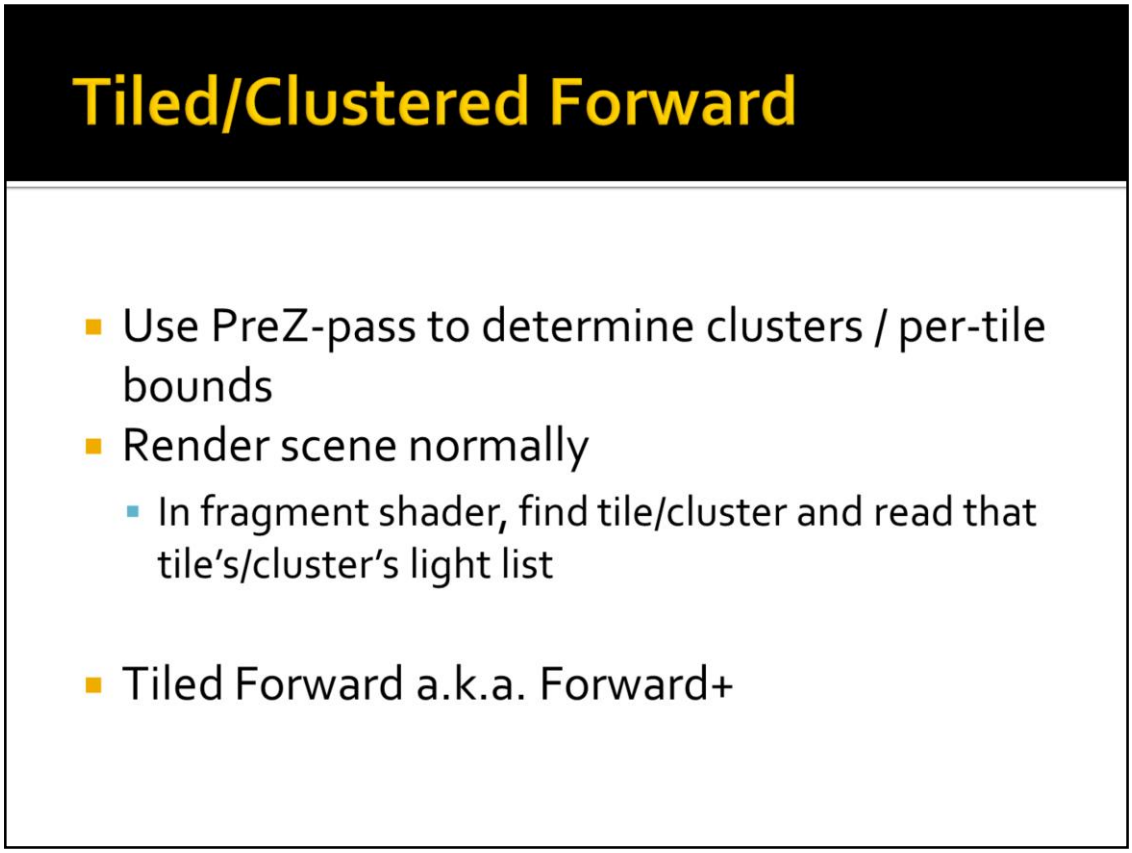
Tiled/Clustered deferred works similarly. The scene is first rendered to G-Buffers. The lighting is them computed using a single full screen pass. The advantage here is that the G-Buffer is read once for each pixel, and the framebuffer is written once per pixel too. This is a fair improvement in terms of memory bandwidth compared to the traditional deferred method already.



However, G-Buffers are still used, so support for multiple render targets is still an issue.

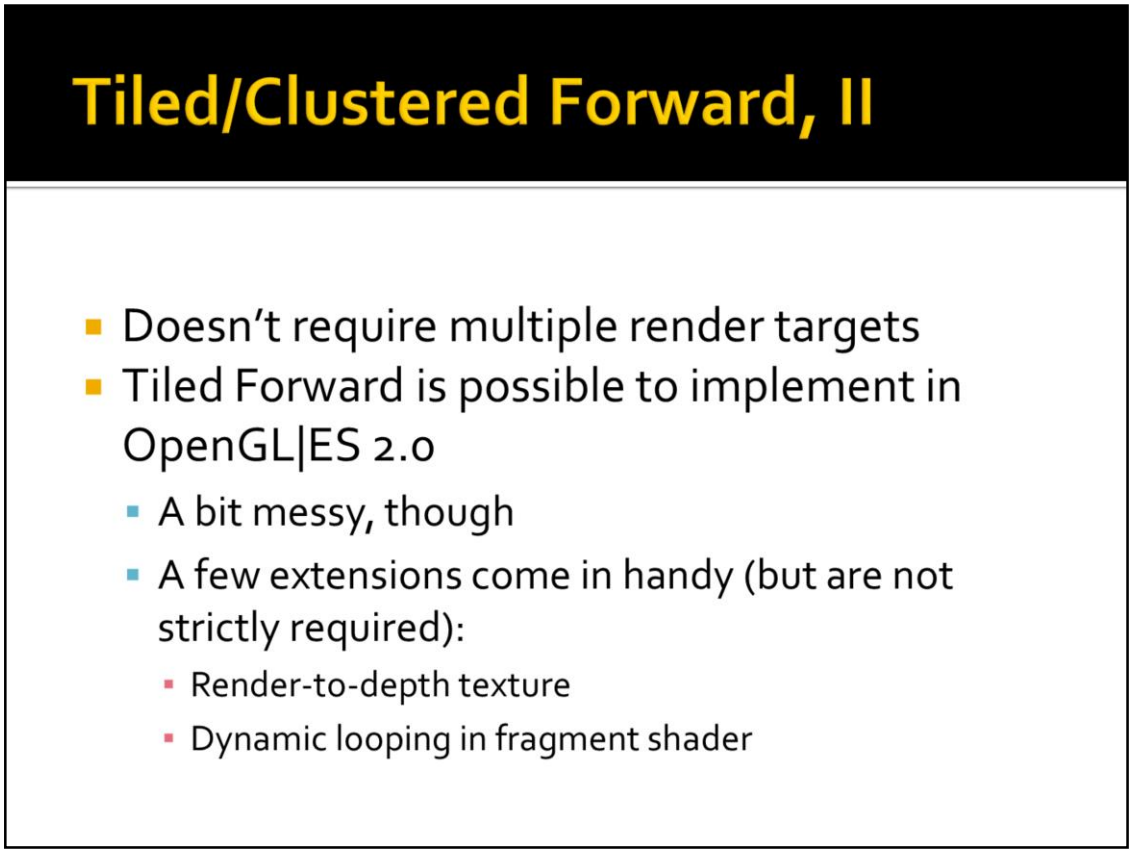
Our original method also heavily relies on compute shaders in order to identify valid clusters and for light assignment to these clusters. For clustering, this is an issue. Extracting the valid clusters from the depth buffer should be especially tricky without compute shaders that can access the depth buffer directly.

For tiled shading it's possible to compute each tile's depth bounds using a fragment shader and then perform the light assignment on the CPU. This requires a read-back to the CPU however.



The original Tiled/Clustered forward method uses a PreZ pass to determine clusters or per-tile bounds. This has issues similar to the deferred version I just discussed – however, again, for tiling it's perfectly doable.

Afterwards, the geometry is rendered normally. Here, in the fragment shader, we'll find which cluster or tile the fragment is a part of and then read that tile's or cluster's light list.



Tiled/Clustered forward avoids the G-Buffers and therefore doesn't require multiple render targets.

As already mentioned, Tiled Forward (a.k.a. Forward+) is possible to implement in OpenGL|ES 2.0. It's a bit messy though, especially if you lack extensions like render-to-depth texture or don't have support for dynamic looping in the fragment shader. There's also a fair bit of packing depth values into RGBA8 buffers, so there are some precision issues that can noticeably impact the quality of the light assignment.

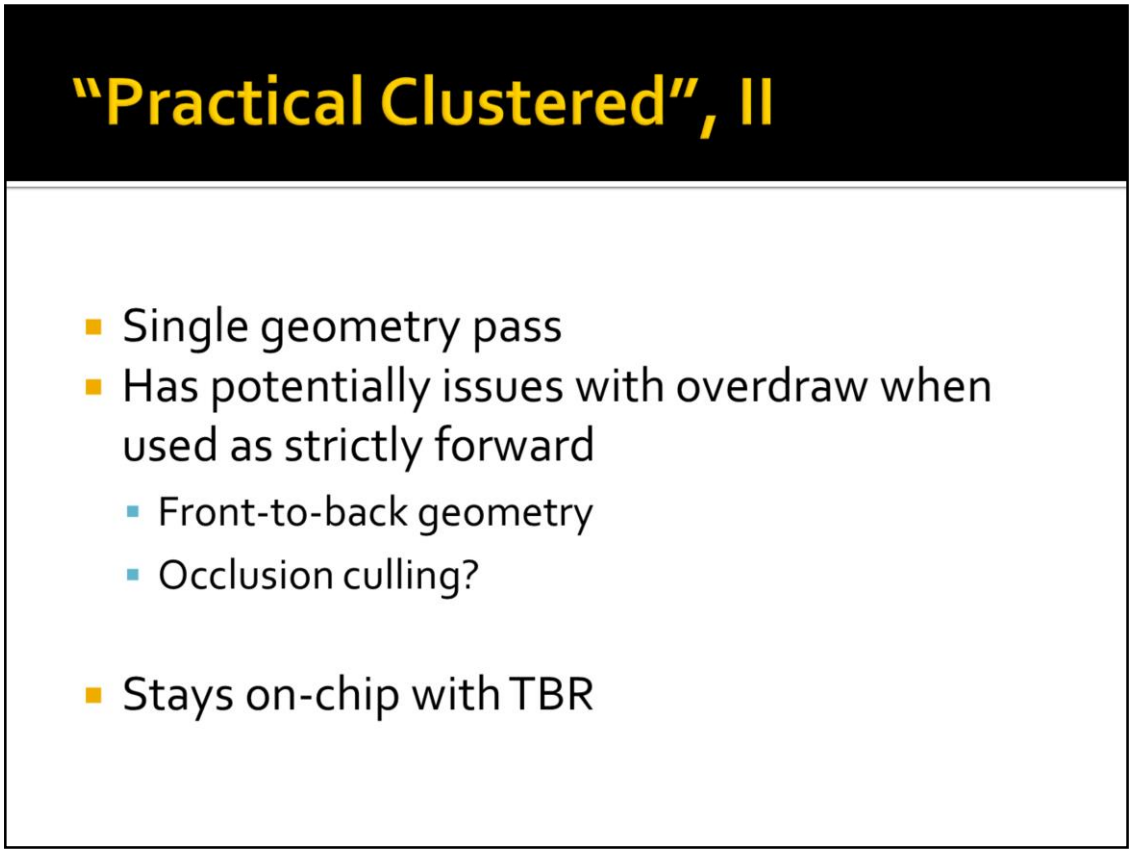
In my implementation of this, I performed light assignment on the CPU. This means that there's a round-trip from the GPU to the CPU and back to the GPU each frame, which has a lot of potential to introduce stalls in the rendering pipeline.

"Practical Clustered"

- Clustered variant as presented by Emil
  - Up-front light assignment to dense cluster structure on CPU
- Applicable to both deferred and forward
  - Or a mix of them…
  - I'll mostly refer to the forward variant, though

Next up is the practical clustered variant presented by Emil in this course. Here the light assignment is done up-front into a dense cluster structure on the CPU. With this, it's possible to avoid a PreZ pass to find the clusters.

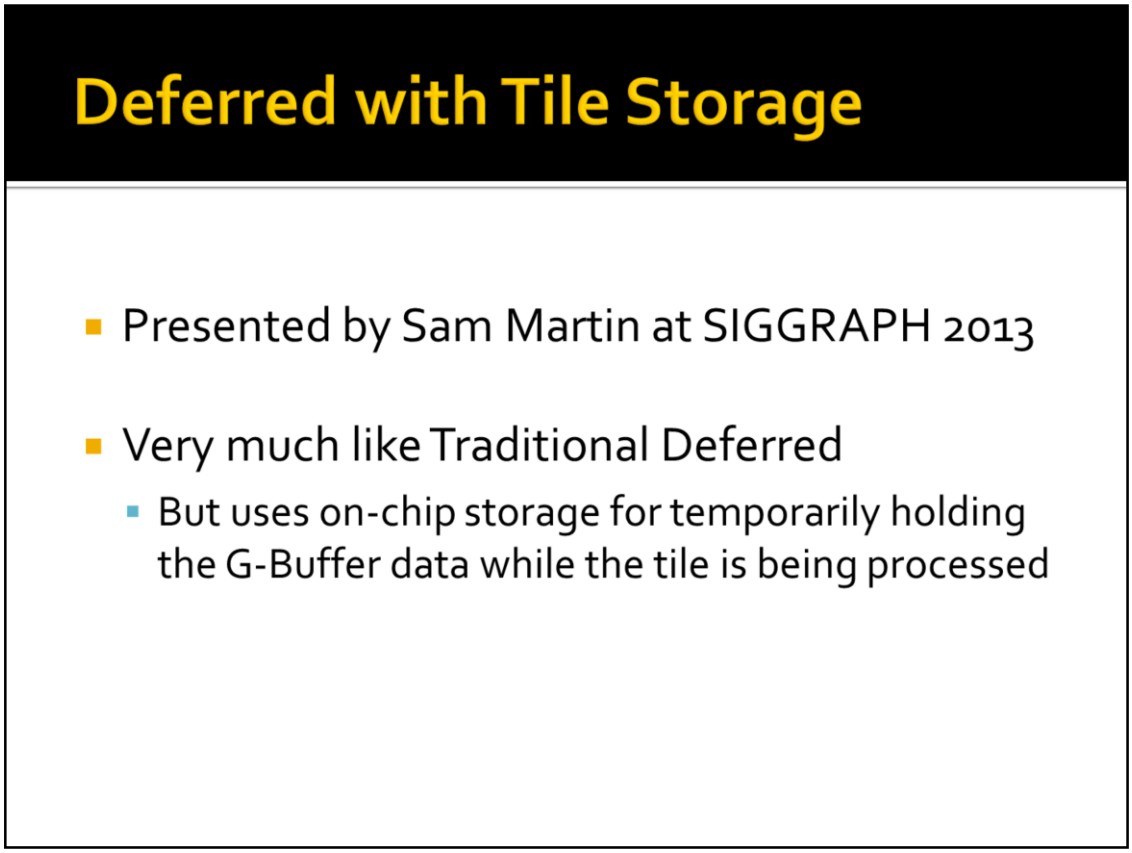
As mentioned by Emil, this method is applicable to both deferred and forward, or a combination of both. I'll mostly refer to the forward variant, though, since that works better on TBR platforms and avoids the need for MRT.



The practical clustering can work with a single geometry pass. However, if used in a strictly forward setting, there's a lot of potential for overdraw and overshading. This can be mitigated somewhat by e.g. rendering front-to-back and/or by employing occlusion culling and the likes.

The main feature here is that each tile's framebuffer contents can stay on-chip for the whole frame.

This should further be really nice on tile based deferred platforms which should avoid all issues with overdraw.



The final two methods were presented by Sam Martin from Geomerics and his colleagues at e.g. SIGGRAPH 2013.

The first of these methods looks a lot like traditional deferred, but it uses the on-chip storage to temporarily store each tile's G-Buffer data while that tile is being processed. The G-Buffer data is never transferred off-chip – only the final colors are stored.

Deferred with Tile Storage, II

- Relies on OpenGL Extensions
  - EXT_shader_pixel_local_storage
  - ARM_shader_framebuffer_fetch
  - ARM_shader_framebuffer_fetch_depth_stencil
- Supported by e.g. ARM Mali T604
  - But extensions unavailable on e.g. the Nexus 10

This method – both his methods in fact – rely on a few OpenGL Extensions. Specifically, there's the EXT_shader_pixel_local_storage which makes it possible to store the G-Buffer data in the on-chip memory without associated render targets.

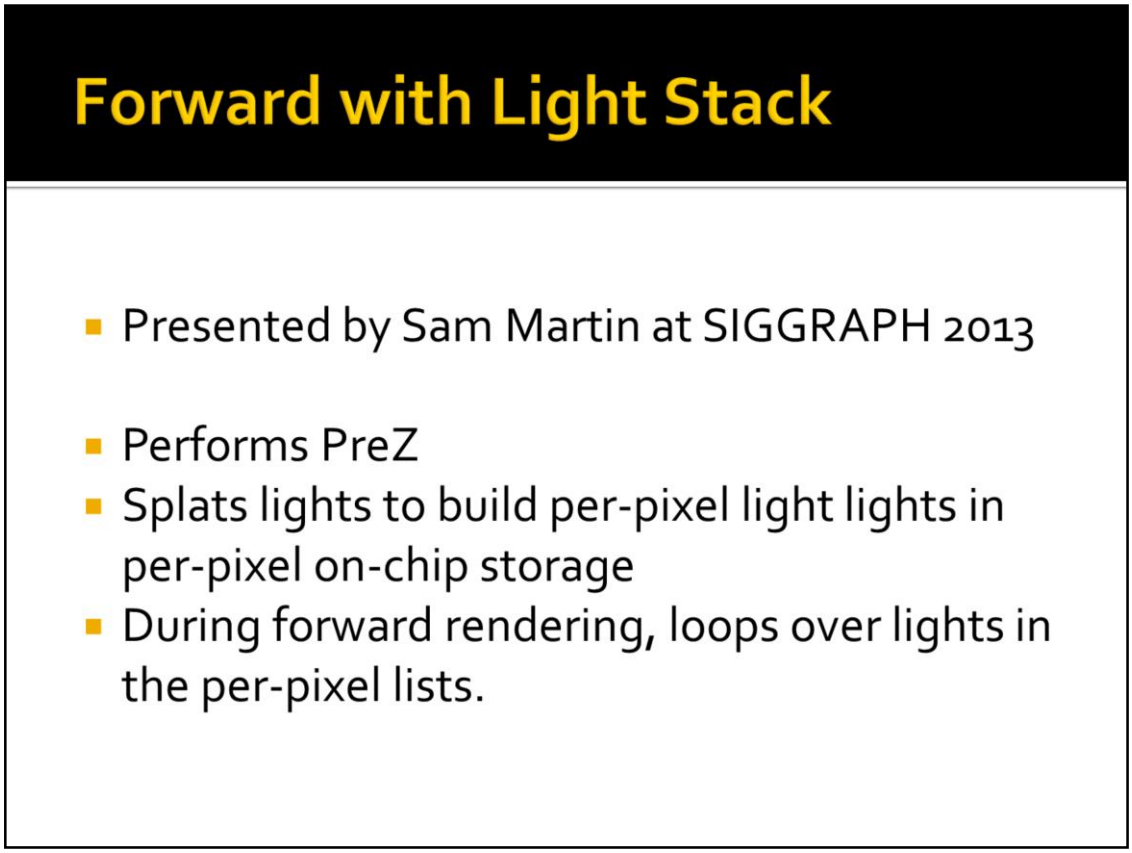
The extensions are, in theory, supported by the ARM Mali T604 hardware. However, at the time of writing all three extensions are unavailable on e.g., the Nexus 10 tablet. In Sam Martin's talk, a dev-board running a "normal" (non-Android) Linux was used.

Deferred with Tile Storage, III

- EXT_shader_pixel_local_storage gives a small amount of on-chip per-pixel storage
  - Preserved across fragment shader instances
  - But not backed by external RAM
- A bit finicky
  - E.g., writes to color outputs destroy the per-pixel storage…

As already hinted at, the EXT_shader_pixel_local_storage extension enables the user to allocate a small amount of on-chip per-pixel storage in the fragment shader. This storage persists across multiple fragment shader invocations, but it's not backed by external RAM.

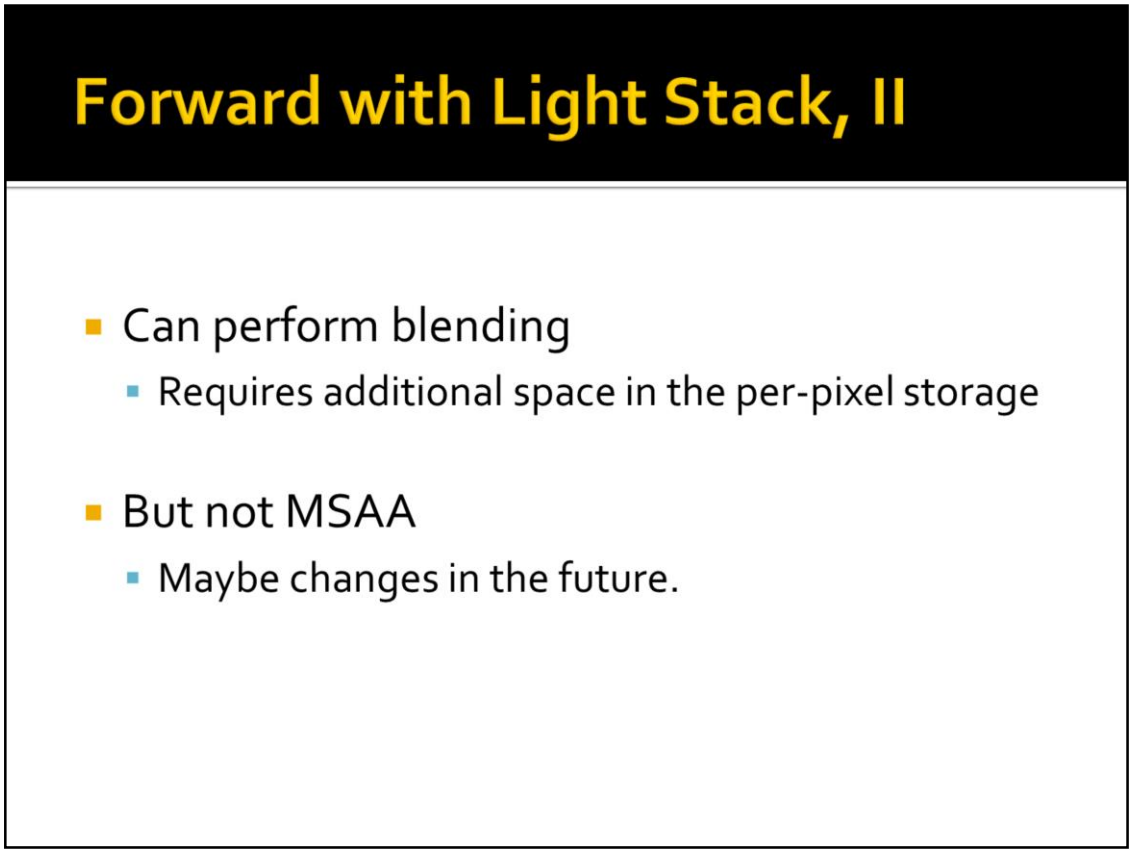
Judging from the spec, the extension seems a bit finicky. There's quite a conditions where the contents of the per-pixel storage are destroyed. For example, writing to a normal color output seems to destroy the per-pixel storage. If you decide to try this out (and have hardware that supports the extension), I'd definitively recommend reading the extension spec very carefully.



This is the second method presented in the talk by Sam Martin.

The method works roughly as follows: first, a PreZ pass is performed. Next, lights are splatted by rendering their bounding boxes, similar to the traditional deferred method. Here, however, instead of computing shading, the shader adds light IDs into per-pixel light lists that are stored in the on-chip storage provided by the EXT_shader_pixel_local_storage extension. This creates the per-pixel “light stacks”.

After this light assignment, a forward pass is performed, where the light list are fetched from the on-chip storage for each pixel.



It's apparently possible to implement blending with this – however, since writing to the normal color buffer destroys the per-pixel storage, the blending has to be performed by hand. The results further need to be stored in the per-pixel storage, so additional space must be reserved for this.

MSAA is currently incompatible with the EXT_shader_pixel_local_storage extension, so that's not an option for now.

Comparison - Summary

	Plain Fwd	Trad. Deferred	Clust. Deferred	Clust. Forward	Practical Forward	Deferred Tile Store	Forward LightStack
Off-chip G-Buffer	No	Yes	Yes	No	No	No	No
#geometry passes	1	1	1	2	1	1	2
Overshading IMR/TBIMR	Yes	No	No	No: PreZ	Yes	No	No: PreZ
Supports MSAA(1)	Yes	No	No	Yes(3)	Yes	No	No
Supports Blending(1)	Yes	No	No	Yes	Yes	No	Yes(2)
GL|ES	Any?	3.0/MRT	3.0/MRT	2.0	2.0	3.0 + Exts.	3.0 + Exts.

(1) Out of box; (2) Blending must be done by hand into per-pixel storage
(3) Correctly accounting for MSAA when deriving per-tile bounds can be tricky

Here's a summary of all the methods that I just presented with a few properties.

Plain Fwd included for reference – it typically scales badly with #lights.

The first row says whether or not an off-chip G-Buffer is required. Off-chip G-Buffers additionally imply requiring support for multiple render targets.

Another interesting point is support for MSAA. Normally, devices like tablets and phones have very high-resolution screens, which make rendering in native resolutions somewhat expensive. A common strategy is to render to a lower resolution view that is then upsampled. In my very subjective opinion, I've found that rendering with MSAA can improve visual quality quite a bit when doing this, without necessarily being terribly expensive. So, having support for MSAA is definitively very good feature, in my opinion.

A note on the Tiled Forward / Forward+ relating to MSAA is that you most likely need to account for MSAA when you compute the per-tile bounds. Not doing so will likely result in some visual glitches. Currectly accounting for MSAA here is actually quite difficult in OpenGL|ES 2.0 since it doesn't have multisampled textures.

Practical Clustered Forward

- Next up: implementing Practical Clustered Forward
- Let's quickly look at why I ended up choosing this.

The next and final part of my presentation is about implementing practical clustered forward. Let's quickly look at why I ended up choosing that method.

Practical Clustered Forward

- Initially, the Nexus 10 only supported OpenGL|ES 2.0
  - So, we ended up being limited by that.

When I started my work on mobile rendering during my time in the Visual Computing team at the Bosch Research and Technology Centre, there were basically no OpenGL|ES 3.0 devices available. So this ended up being a bit of a limitation…

Comparison, Revisited

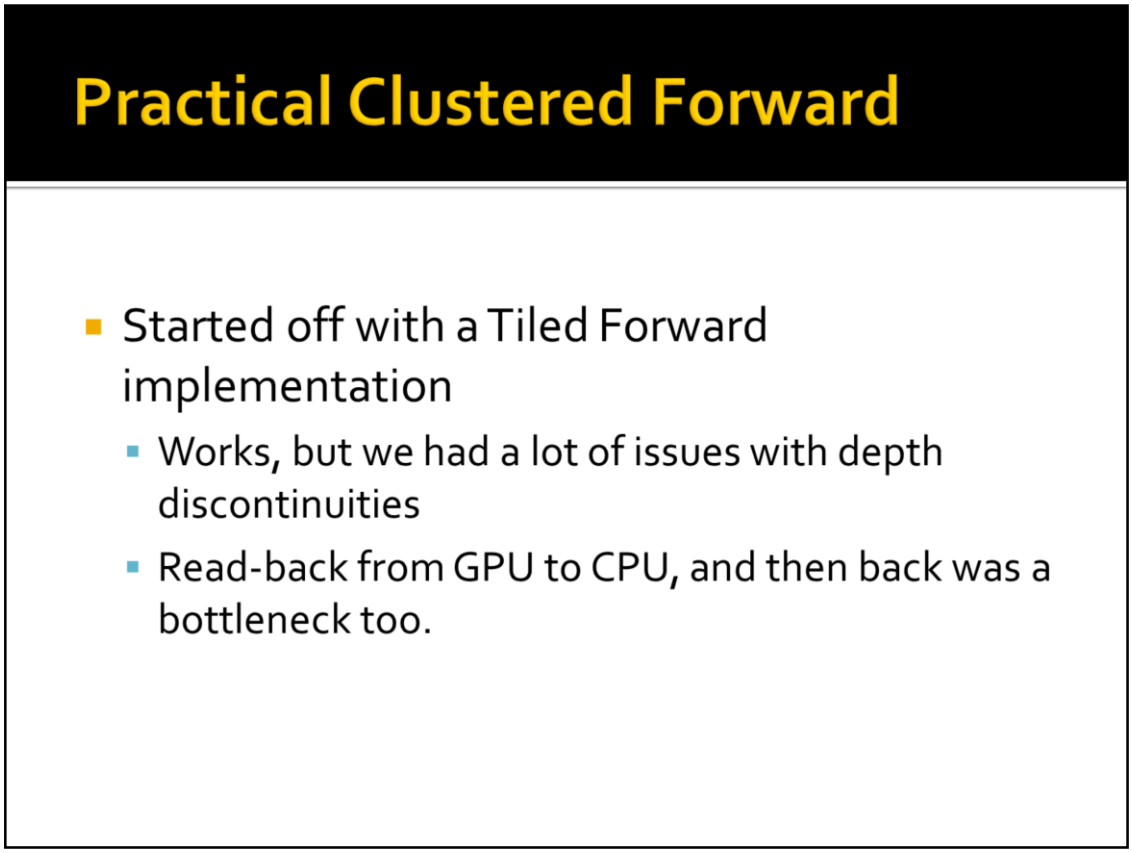
	Plain Fwd	Trad. Deferred	Clust. Deferred	Clust. Forward	Practical Forward	Deferred Tile Store	Forward LightStack
Off-chip G-Buffer	No			No	No		
#geometry passes	1			2	1		
Overshading IMR/TBIMR	Yes	No	No	No: PreZ	Yes	No	o: PreZ
Supports MSAA[1]	Yes		o	Yes	Yes		
Supports Blending[1]	Yes			Yes	Yes		
GL|ES	Any?	3.0/MRT	3.0/MRT	2.0	2.0	3.0 + Exts.	3.0 + Exts.

Without OpenGL|ES 3.0 and MRT.

Technically this occurred before SIGGRAPH 2013, so I didn't know of the last two methods at the time.

I'd very much like to implement them and try them out, but by current development device still doesn't support the required extensions, so this is a bit tricky.

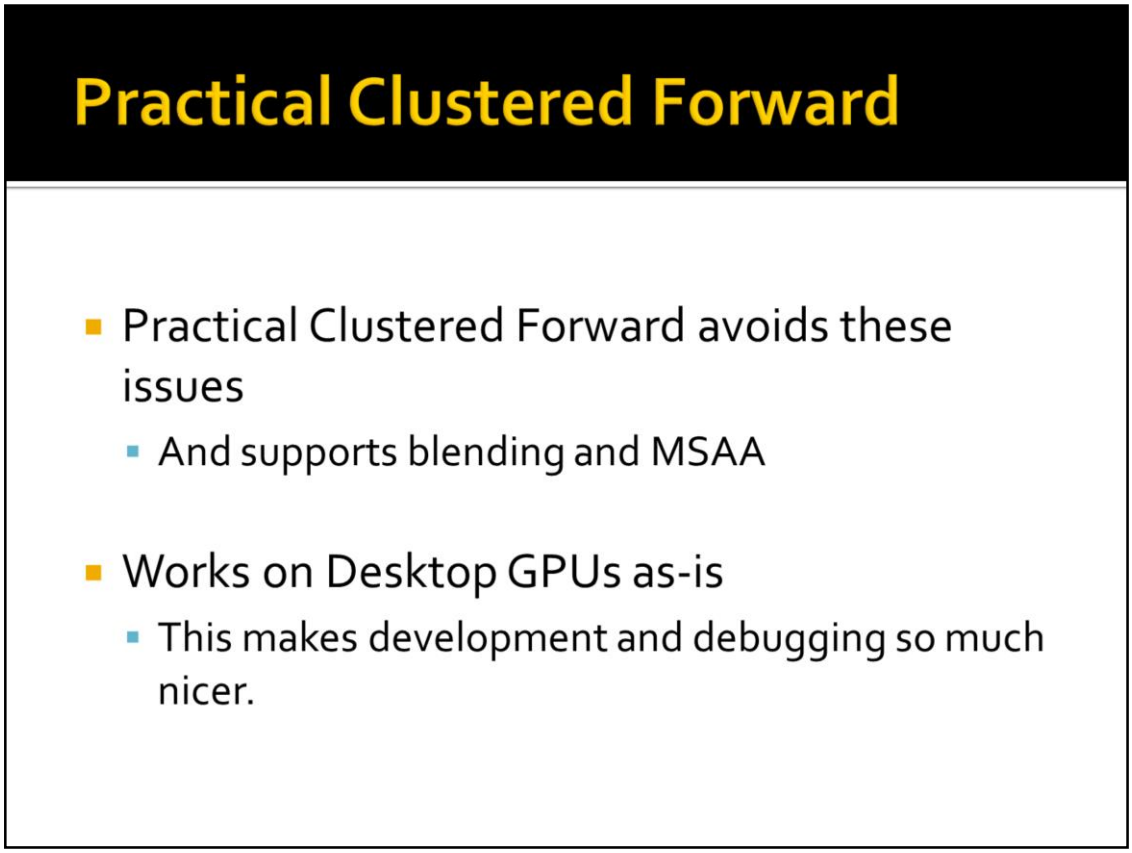
Plain Forward is technically an option, but we wanted to support many lights, so…



I already mentioned this – I stared off with a tiled forward implementation. This is definitively doable. However, we had trouble with the depth discontinuities for a lot of close-up views.

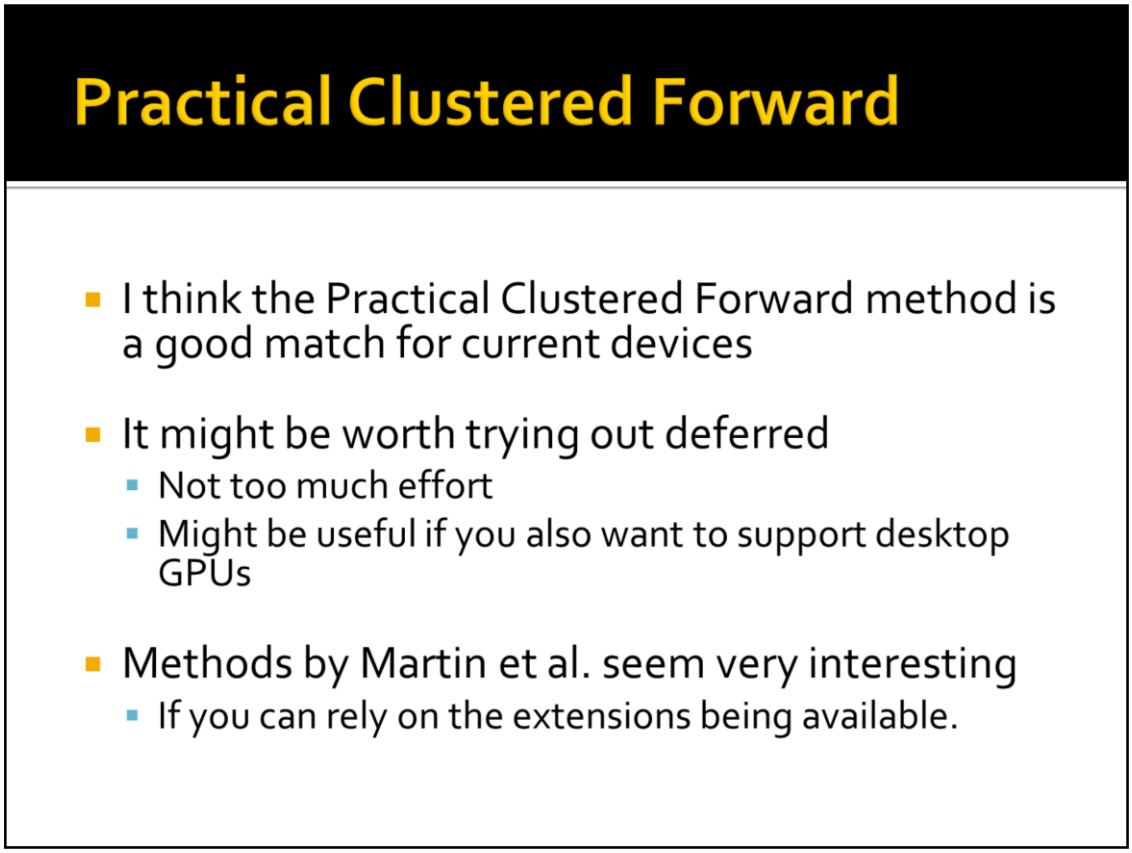
Also, the roundtrip from GPU to CPU and back for light assignment turned out to be a bit of a bottleneck.

The implementation was a bit messy too, since data needed to be packed into various combinations of 8-bit channels in color textures. We experienced some precision issues that impacted light assignment performance badly. Also, MSAA presented some problems.



Practical Clustered Forward largely avoids these issues. It supports both blending and MSAA out of the box. Framebuffer data stays in the on-chip memory. Everything is good™.

Practical clustered forward also runs on the desktop pretty much as-is, since it doesn't rely on special extensions. My desktop and Android implementations share all code (with a few #defines), so I can develop and test stuff on the desktop to a relatively large extent. This makes development and debugging much, much nicer.



For now, I think that Practical Clustered Forward is a good option for many-light rendering on current mobile devices. It has very low requirements, so it should run on a broad selection of different devices.

It can also easily be extended to have a deferred rendering component, which might be useful if you want to support desktop GPUs.

The methods by Sam Martin seem very interesting, but the limited availability of the required extensions might be problematic.

Practical Clustered Forward Implementation

Clustering, Redux

- Seen a couple of different methods of clustering
  - E.g., the original sparse exponential clustering
  - Emil's up front dense clustering
  - (Tiling, which is a special case)
- Going to show one more.

During the course, a few different ways of performing the clustering in Clustered Shading have been shown so far. The original method performs a sparse exponential clustering, where each cluster is kept approximately cubical and with a fixed footprint in the framebuffer. In the sparse clustering only clusters that are occupied are considered, which makes it possible to use a very high resolution setup.

Emil later presented his practical method, which performs the clustering up front into a dense structure.

Tiling was also considered, which could be seen as a special kind of clustering with just one depth slice. :-)

I'm going to present one more clustering method, which is what I've ended up using.

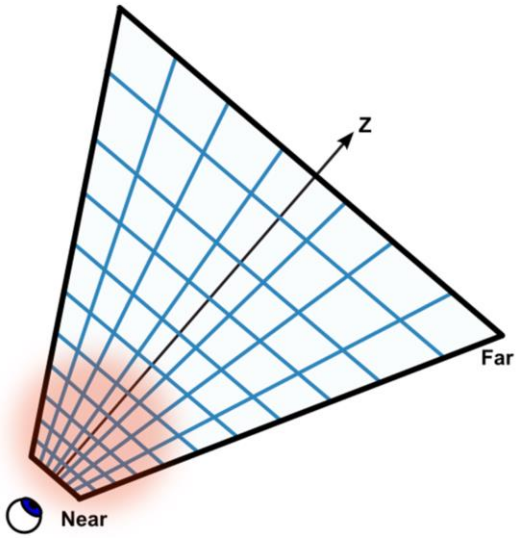
Upfront Clustering – The Problem

- Up front clustering assigns light into a dense structure.
  - Possibly a ton of clusters.
  - The original sparse method avoids this by only considering occupied clusters

Since I'm, like Emil, doing the clustering up front, I have to assign lights into a dense structure. The problem here is that this dense structure can potentially contain a large number of clusters. As mentioned, the original sparse method avoids by only considering occupied clusters. Unfortunately, we can't do this.

Upfront Clustering – The Problem

- Lower resolution?
- Observation: Problematic close to the camera:
  - Lot's of tiny clusters



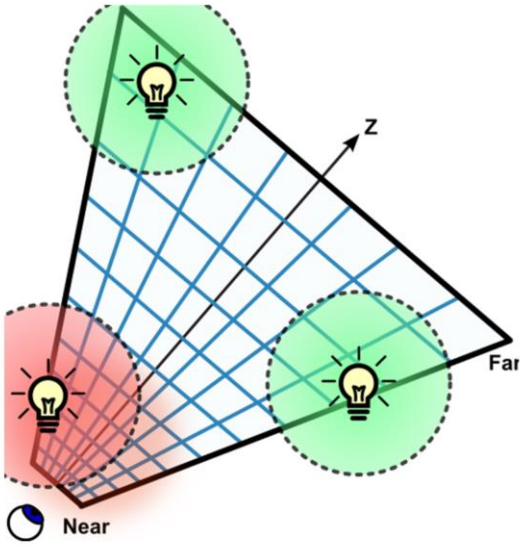
One way to reduce the number of clusters is by lowering the resolution of the clustering.

This reduces the problem with the number of clusters. However, it also reduces the utility of the clustering, since the light-to-pixel mapping becomes more inaccurate. This results in more false positives in the light assignment and end up doing additional, unnecessary work during shading.

An observation here is that this is especially problematic close to the camera with the exponential clustering – close to the camera, there's a whole lot of really tiny clusters.

Upfront Clustering – The Problem

- Lower resolution?
- Observation: Problematic close to the camera:
  - Lot's of tiny clusters
  - A single light source will end up in all of them.



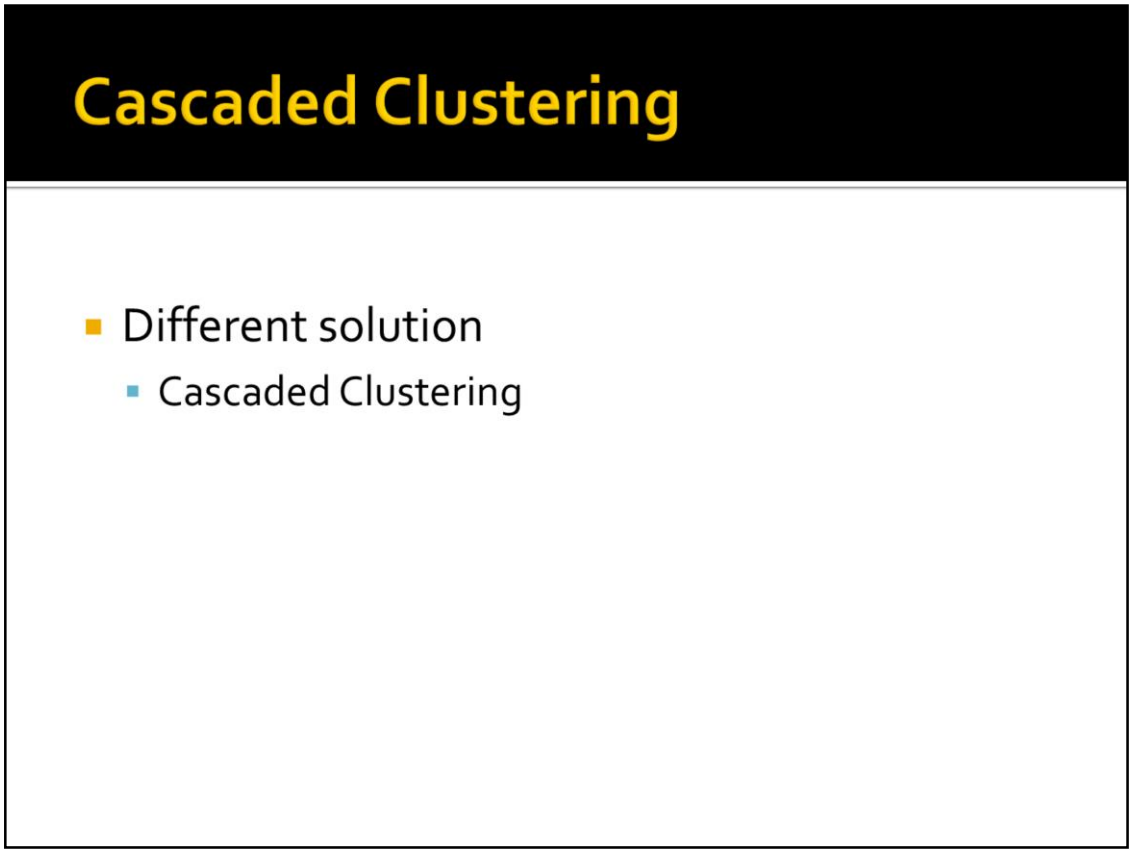
These clusters don't really contribute a lot. Light sources close to the camera will simply end up in all clusters close to the cameraV

Upfront Clustering – The Problem

- Lot's of extra work during light assignment
  - With little to no gain
  - Especially problematic if camera moves through light volumes
- Discussed previously
  - E.g., move first subdivision back as shown by Emil
  - Still have a lot of slices in the XY plane, though.

So, these tiny clusters contribute a lot of extra work during light assignment – work from which we don't gain anything during rendering. This is especially problematic if the camera is allowed to move through light volumes.

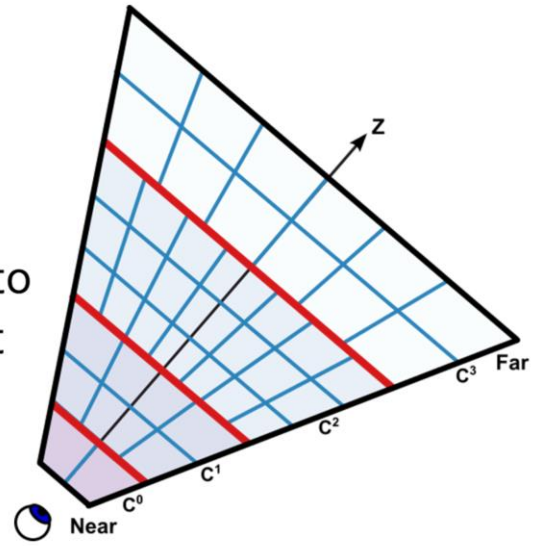
Emil suggested moving the first depth subdivision further back. This certainly reduces the number of clusters close to the camera, but it still leaves a large number of slices in the XY plane.



I've been looking at a different solution: Cascaded Clustering.

Cascaded Clustering

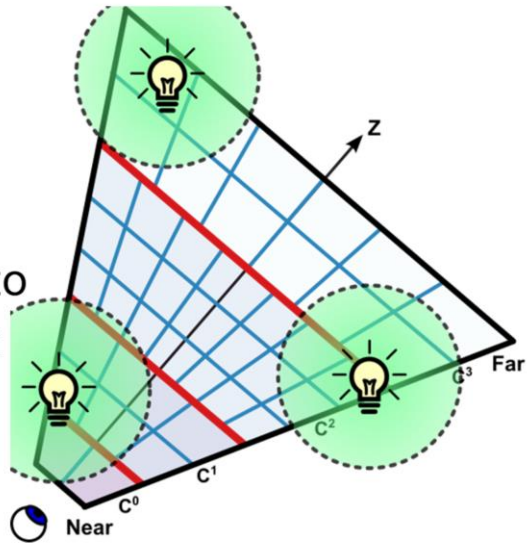
- Different solution
  - Cascaded Clustering
- Subdivide frustum into "cascades" and select individual resolution for each



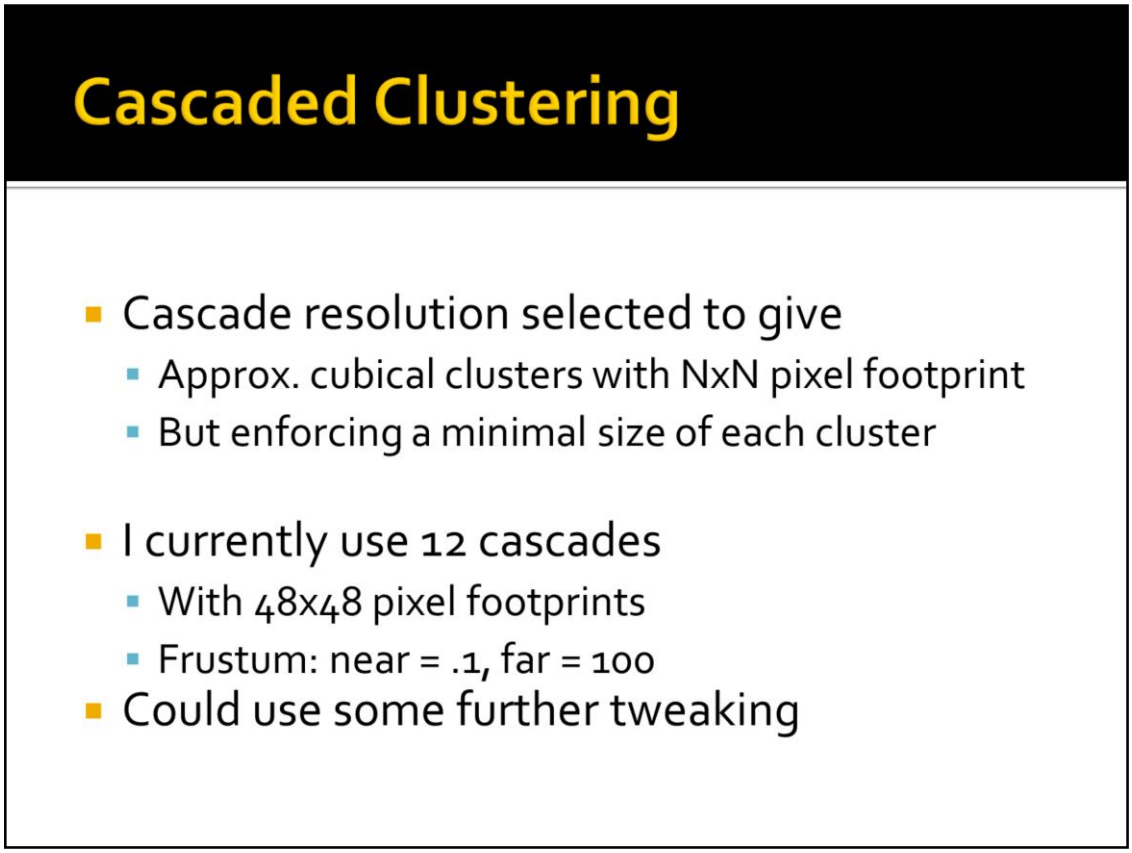
The idea here is the subdivide the frustum into cascades along in the depth direction. We can then select an individual resolution for each cascade.

Cascaded Clustering

- Different solution
  - Cascaded Clustering
- Subdivide frustum into "cascades" and select individual resolution for each



By selecting a lower resolution for cascades closer to the camera, tiny clusters there can be avoided.



Each cascade's resolution is selected to yield approximately cubical clusters with a NxN footprint (as previously). But now I'm also enforcing a minimal size for each cluster.

Currently, I use 12 cascades. The target pixel footprint is 48 by 48. The setup can use some further tweaking, I don't think that's necessarily the optimal setup at this time. But it already performs well enough.

Cascaded Clustering

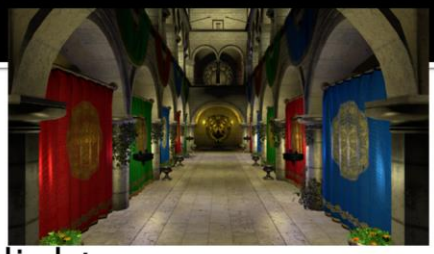
Compute cluster ID in shader:

```
highp float fragCoordZ = 1.0 - 2.0*gl_FragCoord.z;
highp vec2 fragCoordXY = gl_FragCoord.xy * uCascadeData.recWindowSize;

// find current cascade
mediump float Z = uCascadeData.Z.y / (fragCoordZ - uCascadeData.Z.x);
mediump float cascadef = floor(
    log(Z / uCascadeData.cameraNear) / uCascadeData.logOnePlus2TanFov
);

lowp int C = int(cascadef);

// compute cluster id
highp vec3 clusterXYZ;
clusterXYZ.xy = fragCoordXY * uCascadeData.params[C].xy;
clusterXYZ.z = log(Z / uCascadeData.params[C].z) / uCascadeData.params[C].w;

highp vec3 clusterXYZfloor = floor(clusterXYZ);

highp float clusterf =
    clusterXYZfloor.z * uCascadeData.params[C].x * uCascadeData.params[C].y
    + clusterXYZfloor.y * uCascadeData.params[C].x
    + clusterXYZfloor.x
    + uCascadeData.offsets[C];
```

Looking up a cluster becomes a two step procedure: first the cascade index of the fragment is computed. Using this, the cascade's parameters (such as resolution) can be computed. The second part is more or less identical to the computations for a simple exponential clustering – each cascade behaves just like a frustum with some special near and far planes.

Clustering - Results

Clustering - Results



- 65 light sources
- Approx. 6000 non-empty clusters with on average 4 light sources
- Worst cluster has 14 lights

- Clustering takes approx. 0.75 ms by itself
  - On the Nexus 10 tablet
  - For comparison: without cascades this takes around 5ms with 40k non-empty clusters

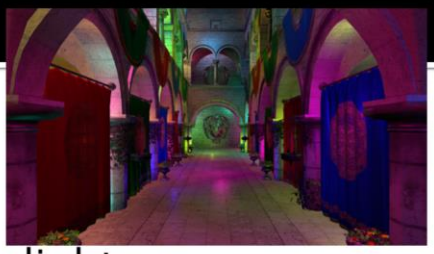
This Sponza contains 65 light sources. The cascaded clustering I described produces approximately 6000 non-empty clusters. The non-empty clusters contain 4 light sources on average. The worst-case cluster contains 14.

Performing the clustering takes less than a millisecond on the Nexus 10 tablet. This time excludes things like uploading the updated cluster data to the GPU, however.

For comparison, performing the clustering without the cascades takes around 5ms and produces 40k non-empty clusters. Both cases target 48x48 pixel footprint clusters. In the cascaded case, some of the clusters close to the camera obviously end up having a larger pixel footprint, though.

Clustering - Results

Clustering - Results



- 192 random light sources
- Approx. 9000 non-empty clusters with on average 14 light sources
- Worst cluster has 38 lights

- Clustering takes approx. 2.5 ms by itself
  - Again on the Nexus 10 tablet

Here I've increased the number of lights slightly, to 192 light sources. With the cascaded clustering, this produces approximately 9000 non-empty clusters. Non-empty clusters contain on average 14 light sources now. The worst cluster has a grand total of 38 lights in this view.

Just computing the clustering takes approx. 2.5 ms. This is again measured on the Nexus 10 tablet.

Results

- Average around 40 ms per frame total time
  - 1280 x 720
- Worst case: 60 ms per frame
  - E.g., view to the left
- Use PreZ due to unsorted geometry.
  - I.e. two geometry passes.

Total rendering speed is around 40 ms per frame for 720p. The worst case is a bit slower at around 60 ms per frame. The view to the left that I've shown is pretty close to the worst case: a large part of the scene is visible.

An interesting note is that I perform two geometry passes here. I ended up enabling the PreZ pass, since it ended up improving performance noticeably. This is probably related to the fact that the geometry is submitted willy-nilly, in some random order. There's also no culling of any kind going on.

To conclude…

Conclusion

- There exist many light rendering methods that are viable on mobile hardware.
- The "Practical Clustered Forward" seems like a good choice at this point
  - I may be a bit biased, though
- Other methods with very good potential out there (e.g. On-Chip Deferred)
  - Especially if they become available on "normal" devices ;-)

References

- Challenges with High Quality Mobile Graphics, Martin et al. 2013
- The Revolution in Mobile Game Graphics, Martin and Bjørge 2014
- An Evolution of Mobile Graphics, Shebanow 2013
- Forward+: Bringing deferred lighting to the next level, Harada et al. 2012
- Our work on Tiled / Clustered Shading:
  - Clustered Deferred and Forward Shading, Olsson et al. 2012
  - Tiled and Clustered Forward Shading, Olsson et al. 2012
  - Tiled Forward Shading, Billeter et al. 2013